University of Southern Queensland
Faculty of Engineering & Surveying

Remote Access of Automated Test Equipment

A dissertation submitted by

Nathan John Hetherington

in fulfillment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering (Computer Systems) &
Bachelor of Information Technology (Applied Computer Science)

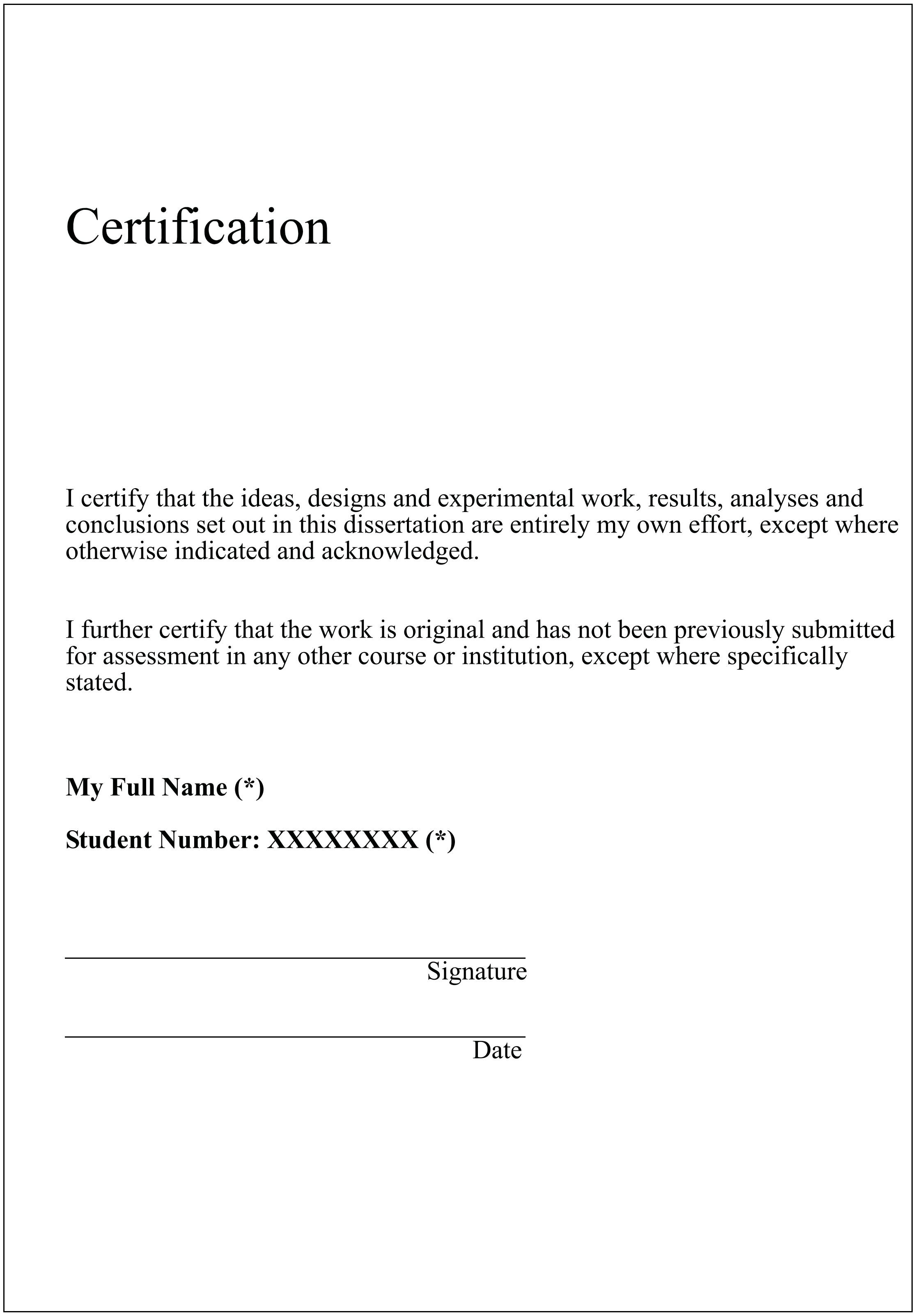
Submitted: October,2004

Abstract

The control of Programmable Instruments using the GPIB, is possible through the use of Application Programmers Interface (API). This thesis investigates the possibilities of using a web server to provide remote access, to allow interactive control of the instruments in an Automated Test Equipment(ATE) rack.

Fully documented programs where developed using the lcc-win32 software development environment, and tested using an Apache web server. Programs and web pages where developed for all the instruments included in the ATE Rack.

A web server to host all the pages developed for interaction with the ATE rack was configured to provide a password controlled system. But due to security issues regarding the PC that the equipment is attached to, connection to the Internet was not possible. But some tests where performed to simulate the various components that the Internet is comprised of, to detect possible errors that may occur.



Certification

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

My Full Name (*)

Student Number: XXXXXXXX (*)

Signature

Date

Figure 11.3: The required text and layout of the Certification page of your dissertation. Substitute your name and student number as indicated (*) and **reproduce the text exactly as shown here**.

Acknowledgements

The assistance received by continuous meeting with one of my supervisor David Parsons through the duration of the project, aiding in the various aspects of the ATE equipment and the direction of this project. Also for acquiring and passing on of information relevant to this project.

The assistance received by supervisor John Leis with regards to the different aspect of this project and possible solutions.

The assistance received by laboratory technician Terry , who helped with gaining access to the ATE equipment and assistance when problems where encountered.

My partner Rebecca who tolerated me through this stressful time, and got me coffee.

Contents

1 Introduction **1**

2 IEEE 488 General Purpose Interface Bus **3**

2.1 History of IEEE 488 Standard 3

2.2 Type of IEEE 488 Cards Available 4

2.3 IEEE 488 Connector and Wire Configuration 5

2.3.1 Data Bus 6

2.3.2 Transfer Bus 6

2.3.3 Management Bus 7

2.4 IEEE 488 Connection Details 7

3 Application Programmers Interface Libraries **11**

3.1 Standard Instrument Control Library 11

3.1.1 SICL Commands 12

3.2 Virtual Instrument Standard Architecture 14

4 Remote Access Implementation Method **17**

4.1 Project Background 17

4.2 Project Implementation Outline 18

5 Writing C Programs **21**

5.1 Data Types and Structures 22

5.1.1 Basic Data Types 22

5.1.2 Data Arrays 25
5.1.3 Structures 26
5.2 Variable Scope 27
5.3 Input/Output in C 28
5.3.1 Standard I/O with keyboard and monitor 28
5.3.2 Using files for I/O 30
5.4 Functions 31

6 Common Gateway Interface 33
6.1 HTML CGI Program Calling Method 34
6.2 CGI Program Basic Structure 34
6.2.1 Obtaining Input 35
6.2.2 Processing Data 36
6.2.3 Output from CGI Application 37

7 Writing CGI Programs for Devices 39
7.1 Step 1: Open a Session with the Device 39
7.2 Step 2: Getting Data from Input String 40
7.3 Step 3: Obtain Device Commands 40
7.4 Step 4: Changing Settings on the Device 42
7.5 Step 5: Error Detection 42
7.6 Step 6: Output Results 42
7.7 Compiling Device Programs 43

8 Creating Web Pages for Remote Access 45
8.1 HTML Basics 45
8.2 Frames 49
8.3 Creating HTML pages for ATE Instruments 52

9 Setting Up an Apache Web Server 55
9.1 Installing Apache 56

9.2 General Configuration 58
9.3 Configuring Server 59
9.3.1 Controlling Access 59
9.4 Adding Web Pages and CGI Programs 61

10 System Test 63

11 Conclusion 65

11.1 Suggested Further Development 65

A Project Specification 69

B ASCII Character Set 71

C GPIB Interface Cards 73

D HTML Pages 85

D.1 Function Generator 85
D.2 Switch Unit 86
D.3 Power Supply 87
D.4 Oscilloscope 88

E CGI Applications 91

E.1 Basic CGI Applications 91
E.1.1 CGI get Method 91
E.1.2 CGI post Method 97
E.2 CGI Programs for Devices 100
E.2.1 Function Generator 100
E.2.2 Oscilloscope 107
E.2.3 Switch Unit 113
E.2.4 Power Supply 121

F Basic C Programs 127

F.1 Data Types . . . 127
F.2 Structures . . . 128
F.3 Variable Scope . . . 129
F.4 Input/Output Example . . . 129

G Application Interface Library **131**
G.1 VISA Library . . . 131
G.2 SICL Library . . . 255

H Device Manuals **345**
H.1 Function Generator . . . 345
H.1.1 Quick Reference . . . 345
H.2 Oscilloscope . . . 356
H.3 MUX . . . 413
H.4 Power Supply . . . 434

Chapter 1

Introduction

The objective of this project was to develop a method of providing remote access between a users computer and the devices in the Automated Test Equipment rack. This access method was to be in a manner that overcomes some of the problems with the system that was investigated previously. In completing this project the following steps had to be achieved:

1. Interface card had to be identified
2. Application Programmers Interface Libraries had to be obtained
3. Device command had to be Identified
4. Programs that Interface with device had to be written
5. Web pages for interaction with devices had to be written
6. Web server had to be installed and configured

At the moment there is no system that provides remote access to the Automated Test Equipment, this is due to limitations enforced by the Information Technology Services department of the University. The previous system never evolved further then the setup phase before these limitations where put in place for security reasons. The software that was going to be used was Hewlett Packard Virtual Engineering Environment Version 5 and was the predecessor of the software being used on the PC connected to the Test equipment. Version 5 provided remote access, but was hard to configure and required software ports left open (listening) that are not used by the standard applications permitted by firewalls.

Along with the inherited security problems that the software processed, the Windows 95 Operating system on which the software ran also created a security threat to the whole University network. And to compound the already large deficiencies of the system, it's performance was troublesome and unpredictable. Hence a need for a method that would remove the security problems and hopefully improve the reliability and performance is the foundation of this project.

The major proportion of this project is the software generated to provide the remote access. The complete code for this project is contained on the CD-ROM attached to the back of the thesis. This CD is broken it to the various stages required to implement the remote access that was produced by the project. The directories and the files that they contain are :

1. Libraries : Contains the API library files need to write a program.
2. Manuals : Contains the various device manuals that contain the commands for each device.
3. HTML_files : Contains the Web pages need to interface with the devices.
4. C_files : Contains the cgi programs source files that interface with the devices.
5. Apache : Contains the installation executable for the Apache Web Server.

Chapter 2

IEEE 488 General Purpose Interface Bus

The concept of using Automated Test Equipment for automation of test that are mundane and prone to user errors is becoming more of an widely used technique. The ability to write a program for a test sequence is becoming more attractive, as programs that provide a graphical user interfaces become available. These software packages allow the user to interact with the ATE systems via an interface card, without having to worry about the underlying system. The interface protocol that most ATE systems uses is known as the IEEE 488 standard, and has become a solid foundation for future ATE equipment to build on.

2.1 History of IEEE 488 Standard

The IEEE 488 standard was original started in the late 1960's as the Hewlett Packard Interface Bus(HPIB), this was the protocol that Hewlett Packard originally devised to connect and control devices that they manufactured. With the introduction of programmable devices the need arose for the introduction of a standard by which devices from different manufactures could interfaced was required.

In 1975 the Institute of Electrical and Electronic Engineers (IEEE) published the IEEE 488 standard that detailed the electrical, mechanical and functional specifications of the interface bus. This interface bus was specifically designed for the interaction with programmable instrumentation and was reviewed in 1978 but no major changes where made to the standard. This standard is now in world wide use by the major programmable instrumentation manufactures and known by three different names:

1. Hewlett Packard Interface Bus (HPIB)

2. General Purpose Interface Bus (GPIB)

3. IEEE 488 Bus.

Work on standard continued as the original standard did not outline the syntax and formating conventions for the communications with devices. Finally the new standard was completed and released as the IEEE 488.2, with the original standard being dubbed IEEE 488.1. This standard contained information on Codes, Formats, Protocols, and Common Commands for use with the IEEE 488 standard. The new standard did not replace the original, with most new devices supporting both, instead it provides a reasonable level of compatibility.

Finally in 1990 the IEEE 488 standard was appended to include the Standard Commands for Programmable Instrumentation (SCPI) documents. The Standard Commands for Programmable Instrumentation is the most recent attempt to provide compatibility between Programmable Instrumentation from different manufactures, this is topic of [15] . This standard defines specific commands that each instrument class must obey, if followed this will provide complete compatible and configurable systems among these instruments.

2.2 Type of IEEE 488 Cards Available

Currently on the market there are many manufactures that are producing IEEE 488.1 and 488.2 cards for interfacing with programmable instruments.Some of the major manufactures being National Instruments and Agilent Technologies Inc. Each of which are compatible with the different instrument manufactures due to the standards specifications and each with their own system drivers and API libraries.

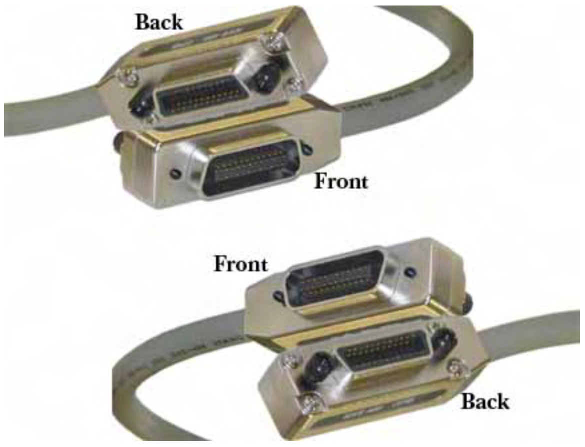
Currently there are a few different interconnection methods between the interface card and programmable Instrument. The main ones being the CN24 connector and cable that uses a 24 pin connector and cable with 24 wires, the RS232 and Universal Serial Bus(USB). The Last two connection methods are not commonly used as they are serial connections, and the ASCII characters used by the IEEE 488 standard is more suited for a Parallel transfer media such as the first method. Appendix C contains the various GPIB interface card produced by Industrial Automation Products and the card used in this project.

The speeds that the serial communication can obtain is less then that of the parallel systems, this is due to the fact that they can only send 1 bit at a time. But the distances that a serial communication link can send is usually larger and the error recovery is easier as the use of a parity bit is often sufficient to correct errors. In parallel communication if an error occurs then more then one bit may be corrupted and retransmit is required. The type of communication link required depends on environment under which the equipment will be used, but the advantages of the 24 wire connection make it the dominant method used.

2.3 IEEE 488 Connector and Wire Configuration

The type of connectors and wiring depends upon the type of interface card that is chosen to interface with the devices. In this section the original 24 wire cable and connector will be considered.The cable and connectors often refereed to as the CN24 connector due to the fact that they are comprised of 24 connector pins and 24 wires, the picture below depicts such cables and connectors:

Figure 2.1: IEEE 488 CN24 cable and connectors



As mention the cable that is usually used for GPIB interactions is a 24 wire cable, of which 8 of these wires are used for ground. The remaining 16 wires are then broken into three groups:

1. Data Bus
2. Transfer Bus

3. Management Bus

2.3.1 Data Bus

The data bus occupies 8 of the 16 signal wires, this is due to the fact that most of the devices use the ASCII character set for there interactions. The ASCII character set only requires a 7-bit number, but most computer systems use 8-bits(1 byte). The advantages of using the ASCII character is that the commands can be recognizable to humans and the data can be manipulated quite easily in programming languages. The data bus is used to transfer data, control information and address.

An example of how the devices use this is shown using the APPL: command used by the Function Generator, It contains 5 ASCII characters that can be sent via the data bus one at a time in binary. So the decimal equivalent of how this command would be sent is 65,80,80,76 and 58. The devices store the characters in a buffer then apply the command, assuming the command is a valid one.

2.3.2 Transfer Bus

The transfer bus provides a handshaking protocol that allows the system to communicate Asynchronously. It does so by allocating each of it's three wire a specific meaning, such as Not Ready for Data, Not Data Accepted and Data Available. Through the use of these control line the system can communication without conflicts and the state of the device can easily be obtain by checking the signals on these wires.

When a device wishes to transmit data on the bus, it sets the DAV line high (data not valid), and checks to see that the NRFD and NDAC lines are both low, and then it puts the data on the data lines. When all the devices that can receive the data are ready, each releases its NRFD (not ready for data) line. Then when the last receiver releases NRFD, and it goes high, the interface takes DAV low indicating that valid data is now on the bus. In response each receiver takes NRFD low again to indicate it is busy and releases NDAC (not data accepted) when it has received the data. When the last receiver has accepted the data, NDAC will go high and the device can set DAV high again to transmit the next byte of data.

This form of handshaking communication allows the devices to transmit at their own pace and provides error detection through the use if the NDAC(not data accepted) line. If any device fails to perform it's part of the handshaking and releases NRFD or NDAC lines then data cannot be sent, this will eventually result in a timeout error.

2.3.3 Management Bus

There are 5 wire allocated to manage the flow of data bytes across the interface and control amongst the various devices connected on the one bus.

The ATN (Attention) signal is asserted by the Interface Card to indicate that it is placing an address or control byte on the data bus. ATN is released to allow the assigned device to place status or data on the data bus. The interface card regains control by reasserting ATN, this is normally done synchronously with the handshake to avoid confusion between control and data bytes.

The EOI (End or Identify) signal has two uses. A device may apply a signal on the EOI line simultaneously with the last byte of data to signal the end-of-data. The interface card can assert EOI along with ATN to initiate a parallel poll. Although many devices do not use parallel poll, all devices should use EOI to end transfers (many currently available devices do not).

The IFC (Interface Clear) signal is set high only by the System Controller in order to initialize all device interfaces to a known state. After releasing IFC, the System Controller is the Active Controller.

The REN (Remote Enable) signal is asserted only by the System Controller. Its assertion does not place devices into remote control mode; REN only enables a device to go into remote mode when addressed to listen. When in remote mode, a device will ignore its local front panel controls until the local button is press on that device.

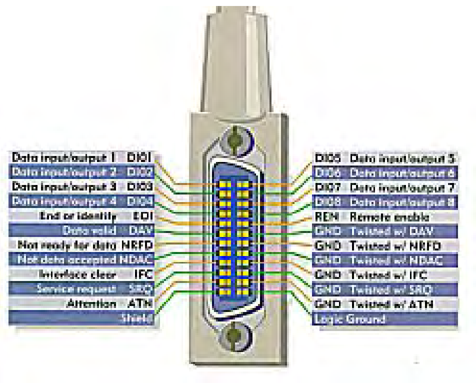
The SRQ (Service Request) line is like an interrupt: it may be asserted by any device to request the interface card to take some action. The interface card must determine which device is signaling SRQ by conducting a serial poll. The requesting device releases SRQ when it is polled.

2.4 IEEE 488 Connection Details

Through the use of the Transfer and Management Bus the IEEE 488 standard is able to have up to 15 devices connected simultaneously, with each device being assigned a unique Primary address ranging from 0-30. This is similar to the way that Ethernet works, the data is broadcast to all devices and the device uses the addressing information to determine if the data was intended for them. A secondary address may also be assigned to each device, also ranging from 0 to 30, but this address is optional and is usually not assigned.

The cables that link the devices is a shielded 24-wire cable, and the distances allowed by the standard is either 20 meters or 2 times the number of instrument connected to the bus, whichever is smaller. The speeds that are achieved by the GPIB standard are in the range of 250 Kbytes/sec to 1MBytes/sec, note it is in Bytes/sec due to the fact that the 8 data bus line. When considering the operating speed of the interface bus the cable size must be taken into consideration, if high speeds are desired the the length on cables used to connect the devices and interface card must be small.
The pin allocation is depicted in the following diagram:

Figure 2.2: IEEE 488 CN24 Connector Pin Configuration



From the diagram and Green[5] it can be seen that the 16 signal wires are broken down as follows :

Data Bus:

DI01 - DI08	Data bus	}	

Management Bus:

		}	Ten
REN	Remote Enable	}	Synchronous
EOI	End or Identify	}	Signals
ATN	Attention	#	

IFC	Interface Clear	#	
SRQ	Service Request	#	Six
Transfer Bus			Asynchronous
NRFD	Not Ready For Data	#	Signals
NDAC	Not Data Accept	#	Twisted Pairs
DAV	Data Available	#	

This shows how the IEEE 488 standard handles the transfer of both synchronous and asynchronous data.

Chapter 3

Application Programmers Interface Libraries

An Application Program Interface as defined by (www.whatis.com,2004) is the specific method prescribed by a computer operating system or by an application program by which a programmer writing an application program can make requests of the operating system or another application. In the case of this project the desired API is for the GPIB card, so that programs can be written to interface with the programmable instruments via the GPIB interface card.

After some researching on the topic of an API for the GPIB interface card contained in the system that the remote access is to be implemented, the manufactures of the card Agilent Technologies Inc provide the API for download as part of a drivers suit. Contained with the drivers which are necessary for correct operation of the card, are two API's. Which where the Standard Instrument Control Libraries and the Virtual Instrument Standard Architecture, and these will be discussed further in the following sections.

3.1 Standard Instrument Control Library

The Standard Instrument Control Library is a modular instrument communications library that works with a variety of computer architectures, I/O interfaces and operating systems according to [14]. This Library allows programs to be written in C/C++ or Visual Basic that can interface with a GPIB card. One of the advantages as stated in [14] is that programs written using this library can be ported at the source code level from one system to another without, or with very few changes.

One down fall of the SICL library is that they are Hewlett Packard dependent, meaning that they can only be used in conjunction with Hewlett Packard hardware. Although it is limiting, the advantage is that the library is efficient,reliable and easy to use with HP hardware. Also the library can be used to communicate over the different interface methods used for programmable instrument communications such as HPIB,GPIB,RS-232 and USB. This library contains a 32-bit and a 16-bit version for windows, which is an advantage as the system connected to the ATE rack is running the Windows©95 operating system which is a 16-bit OS. The SICL library was the only one that worked on the equipment in question for this project, so all programs written will use this library.

3.1.1 SICL Commands

The reader is referred to [14] for a more detailed explanation of the commands and specifics of developing applications using the SICL library, But a brief overview of some of the important aspect of the library will be discussed in the following sections. The concepts discussed in this section are Instrument independent and will be used for the development on CGI programs in a later section. There are a few example programs within [14] so these programs will be referred to rather than new example.

The first thing that an application has to do before it can interface with a device, is create what is known as a device session which creates an ID by which the application can identify the device it is talking too. As mentioned in the IEEE 488 standard each device is given a unique address(see Table 7.1), this address in what is used by the SICL library when communicating with a device. Also when the drivers for the GPIB card are setup, the interface card is given a unique name so that more one interface card can be in the one system at a time, each can be identified by it unique name. The way a session is open is similar to the way that C opens a file and is displayed by the following code:

id = iopen("hpib7,9")

From the line of code above it can be seen that the *iopen* function is used open a session with a device. It only requires the one argument which is a string that identifies the interface card *hpib7* and the address of the device on that interface *9*. If the function is successful then a variable of type INST will be returned and stored in the id variable and will return 0 if it fails, INST is a user defined data type that is defined by the library. This id will be used by the functions that read and write to the devices.

After a session is created then the *iprintf* and *ipromptf* functions can be used to communicate with the device. The strings that these functions use are ASCII characters, and are dependent on the device with which the interaction are with. The use of these two functions are :

> *iprintf(id,"APPL:%s:%s\n",wave_type,VtoV_Peek)*
> *iscanf(id,"%t",buff)*
> *iprompt(id,"*IDN?\n","%t",buff)*

The *iprintf* function can be used with two or three arguments, and follows a similar format to the *fprintf* function for C. The first argument is compulsory as it contains the device's identity that is used in the communication. The second argument is the string that is sent to the device, this string can be a constant string or composed of string variable as long as it follows the format that the device expects. If the second argument contains variable then a third argument must be used and is the list of variable names that correspond to the format identifiers contained in the string.

The *iscanf* function is the similar to the C *scanf* function, except for the addition of the session identifier. The function will read the data on the GPIB and will store it in the variable passed to it, the %t format string specifies that an ASCII string will be read back. The *iprompt* function prompts the device with the string specified and then stores the response from the device in the variable in the fourth argument. The same sequence of events could be achieved by using the *iprintf* followed by a *iscanf*, but this function makes the process neat with only one line required.

Finally after all interactions with the device have been completed then the device session must be close, just like a file in C. This is done using the iclose method passing the session id as it's argument like follows:

> *iclose(id)*

For the purpose of assisting in the writing of reliable code the library provides two functions to add in error detection and correction. The first of which is *ioerror* which installs a default error handling routine which is called if any of the SICL functions result in an error. The second of which is *itimeout* which allows the programmer to specify a time(in milliseconds) that SICL will wait for a response from a device. When incorporated into applications the programmer can detect errors and take action to minimize or fix such errors.

3.2 Virtual Instrument Standard Architecture

The VISA library is similar to SICL, except they where developed with compatibility in mind, allowing the library to be used on any manufactures hardware. 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 as stated in [19].

As with SICL, the VISA must create a device session before any communications between the interface card and programmable instrument can take place. The way that this is done is different then the SICL as two commands are need to create a session. These commands are :

```
viOpenDefaultRM(&defaultRM);
viOpen(defaultRM, "GPIB0::22::INSTR",VI_NULL,VI_NULL,&vi);
```

With the VISA driver a session with the default resource manager must be created before a session with a device. This is due to the fact that VISA is an object-oriented system and the word 'resource' more specifically refers to a particular implementation (or instance in object-oriented terms) of a Resource Class. The resource manager is what maintains all of the class instances, so the class that is about to be create must be registered with it. This class is then linked to the device, this section is similar to SICL. The interface card is identity and the address of the device are specified via a string. The next two arguments are VI_NULL which means "Do not return the number of bytes transferred", these are constants defined in the VISA library which alter the operation of the VISA functions. Finally the last argument is the address of a session variable and is exactly the same as the ID variable in the SICL library, it is used by the I/O functions to determine what devices they are talking too.

The Input and output functions of the VISA library works exactly the same as in SICL. These functions are *viPrintf* and *viScanf* and their arguments are exactly the same as the previously explained *iprintf* and *iscanf*, each requiring a session id, a string and the name of any variables that are being used in the string. The following shows their operation:

```
viPrintf(vi,"*IDN?\n");
viScanf(vi, "%t", buf);
```

Closing the session after all the interactions have taken place is performed via the *viClose* command with the session id being passed to it as it's argument. The VISA library has a large range of error detection functions and are explained in full in [19] and user are referred to this manual if more information is desired.

Chapter 4

Remote Access Implementation Method

With the advances in modern communication systems the potential to provide remote access to application is becoming easier. The Internet has spawned an enormous following and has become a multimedia base communications network. Due to the structure of the Internet it is quite reliable and cheap and for these reasons has become an enormous success. But because it is a shared media and the number of clients cannot always be known, it's performance cannot be measured exactly.

Taking into mind the Internet's inherent advantages and disadvantages this is the media that was in mind during the creation of this project. Nearly everyone can gain access to the Internet from a vast number of locations around the world and due to it's shared nature the cost is relatively low, hence making it a perfect resource. However the shared nature that makes the Internet so appealing causes a problem, security. This problem has manifested itself recently as hackers write viruses, worms and Trojan horses that gain access to systems thought to be secure, causing the need for protection.

4.1 Project Background

Some of the issues mentioned above are the exact reason that the previous attempts to provide remote access to the ATE racks failed. The problem in this case was that the software HP-VEE V5.0 that was being used to provide the remote access created a security hole in the Universities firewall. The way that the software operated required that a software port be left open so the server could listen on this port for any attempts to connect. Hence for correct operation this required that the Firewall did not block this port. As stated by [2] having any open ports exposes you to potential attacks that might exploit known or yet-unknown vulnerabilities, this page also contains links to sites the help identify open ports.

The vulnerabilities that [2] is talking about are those that exist either in the Operating system itself, or Trojan horses that have found their way onto the system. In the case of the system that the HP-VEE software was installed on, the Windows©95 operating system is being used. This operating system was designed when networking and the Internet where not very popular, so the concept of security was over looked or undermined. So as well as the software leaving a port open, the OS on which it run could also jeopardize the security of the whole university network.

With this in mind one way around the firewall would be to use an application that uses a port that is allowed by the firewall. That is exactly why for this project the web server based system was chosen as this application uses port 80 (HTTP) which is allowed by nearly all firewalls. Also most web servers have the ability to specify the files that can be access by a user and also can implement password access. The implementation is similar to that used by most web pages that obtain data and process it in some manner, such as a database searches that the google search engine performs.

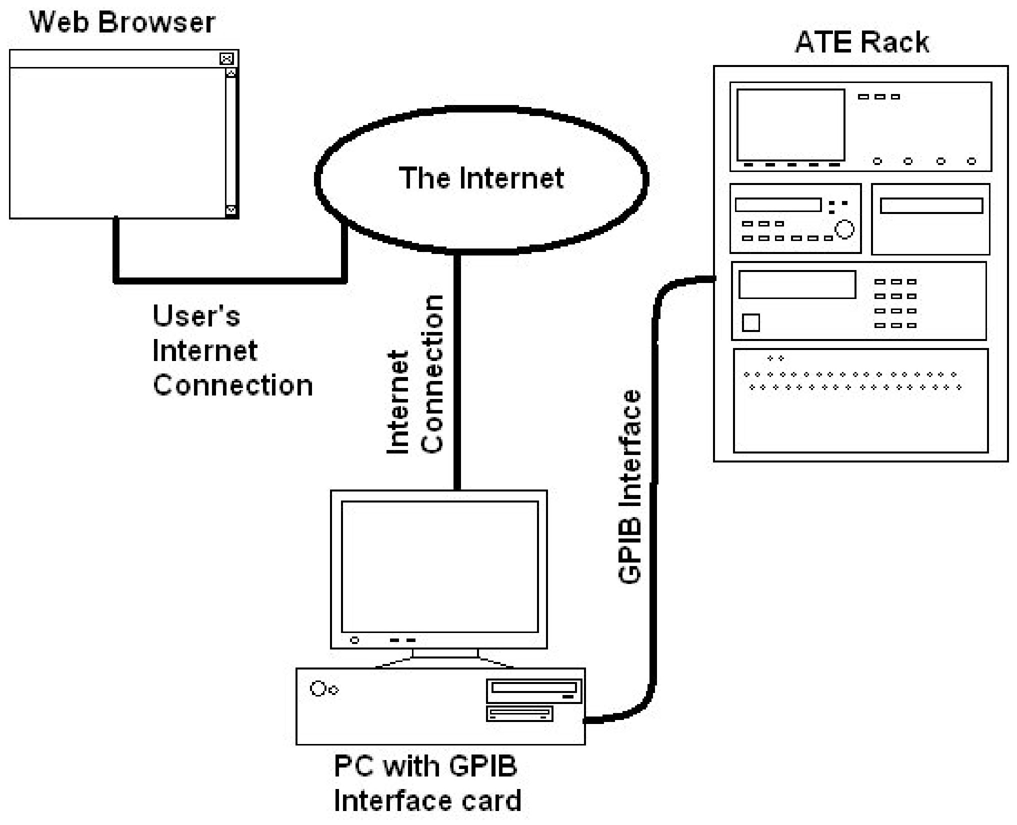
4.2 Project Implementation Outline

The concept of the way that the remote access will be implemented for the ATE rack, is the Common Gate Way Interface Method which will be discussed in detail in chapter 6. But for now the diagram below shows the basic configuration of this method:

The diagram shows the communication paths between the different elements of this implementation. At the centre of the system is the PC with a GPIB card, this system could possibly contain multiple interface cards, and is a detrimental part of the implementation. The PC is responsible for the communications between the user via the Internet and the ATE rack via the GPIB card, and is responsible for controlling and monitoring the access to the services that it provides.

The way that the PC performs these tasks is through an Apache Web Server that is installed and configured to provide the services described previously. All configuration changes can be done via configuration file and are discussed in chapter 9. The web server allows the user to access web pages for interacting with the ATE systems, and these web

Figure 4.1: Implementation of Remote Access using Web Server



pages allow the user to enter data to change settings on the programmable instrument. Once the data is retrieved by the web server, it is passed onto an application.

This application then uses this data to determine the changes to the Instrument that the user desires. The program uses the SICL API to communicate with the device and the desired setting are enforced. For the purpose of detecting problems, the device is queried and the results of this query are check against the data sent by the user. The application is then responsible for replying to the user, and must do so in the form of a HTML code. This response will be displayed to the user as a web page and will inform the user if their request was successful or not.

Because of the modular structure of this implementation, the division of devices can be done using separate web pages for each device and even for the different device sub commands. The way that this is done is up to implementor and is easily changed, making the system easily maintainable. Each web page has one or more corresponding CGI program that is responsible for carrying out the changes on the programmable instrument that it corresponds to. If the ATE rack is upgrade or a device is replace, then it is possible to alter the system with minimal effort.

Also the addition of providing advanced features, such as sequencing, this could be done via separate web pages with their corresponding CGI program. Each page would handle one or more of the steps for the sequence. The actual way that this is implemented is entirely up to the programmer, and due to the modular structure, the separate components could be used separately or in various sequences.

Chapter 5

Writing C Programs

The first and often most important decision when undertaking a software development project is what programming language to use. Some of the criteria that must be considered, as outline in [16], these are Readability, Writability, Reliability and Cost. Each criteria must be considered in great detail as their weighting will vary for different projects, for example a real-time system will place greater importance in the reliability.

Also in conjunction with these criteria, other factors can greatly influence the decision of what programming language to use. In the case of this project, the availability of the Application Programmers Interface for a programming language played a major roll. The API for the GPIB interface card was only available for C/C++ and Visual Basic, so a decision between the two had to be made.

Visual Basic is a simple programming language that allow the user to write programs quick and easily. But as with nearly everything this advantage comes with a cost, visual basic programs can be unreliable and can be limiting in what features are provide for the programmer to use.

The C/C++ programming language on the other hand is one with a much longer history and although it is a high-level language, it also provides low-level features that the programmer can implement into there code. C/C++ are languages that are very powerful and provide the programmer with a large array of features, but can be dangerous if used incorrectly by the programmer.

Although I have had experience with both of these languages, the C/C++ programming language is the one that was chosen for this project. This is due to the fact that I have had more experience with the language and the programs compiled in C/C++

are more efficient (eg. Run faster, more reliable) than most other languages. In addition the advantage of object-oriented programming provided by C++ allows for more efficient code to be produced if desired.

The reader is referred to suitable introductory books if the fundamentals of C/C++ are not known. Suitable books include "Beginning with C" by Ron House [6] and Object-Oriented Programming in C++ by Richard Johnsonbaugh and Martin Kalin [12], however the some of the basics will be outlined in the following chapter.

5.1 Data Types and Structures

The most important structure in any programming language is the variables in which data can be stored, manipulated and retrieved. The C/C++ language has a vast variety of data types that the user can user and structures that allows the user to group variables that logically belong in one structure.

5.1.1 Basic Data Types

Within C there are a few data types and the user can define their own, each of which are used to associate the data stored with it's meaning and method of processing. A table below shows the various data types, the amount of memory they occupy and the type of data stored in them.

There are three basic data types:

1. char - ASCII characters.
2. int - Integer numbers.
3. float - Floating point numbers.
4. Pointers - Stores memory addresses.

The type of data that can be allocated to each variable is enforce by the compiler, so that unintentional assignments don't result data loss and hard to find bugs. The use can force the allocation of a variable into a different type via type casting, this is done by placing a (data_type) in front of the variable to be converted.

For example :

int_var = (int)float_var;

This will cause the floating point variable to be converted to a integer variable and assigned to int_var.

Data Types	**Num. of Bytes**	**Variable Example**
char	1	a,g,b,A,G
int	4	1,2,7,8,12
float	4	1.2,20.4,100.3

Table 5.1: Basic Data Types and Sizes

These data types can only hold a certain range of values, for integers this is -2,147,483,648 to +2,147,483,647. If this range is to short for the desired variable then a modifier, that increases or decreases the number of bytes used by the data type, can be used. There is four of these modifiers available in C/C++, these are :

1. short
2. long
3. signed
4. unsigned

The first two of these modifiers change the number of bytes allocated to the data type it is used in conjunction with. The last two deal with weather or not the most significant bit is used as the sign bit, if it is 1 the number is negative and 0 is positive.

The following table shows the ranges of variables if the modifier is used :

Data Types	**Num. of Bytes**	**Range**
short int	2	-32,768 to +32,768
unsigned short int	2	0 to +65,535
unsigned int	4	0 to +4,294,967,295
int	4	-2,147,483,648 to +2,147,483,648
long int	4	-2,147,483,648 to +2,147,483,648
signed char	1	-128 to +127
unsigned char	1	0 to +255
float	4	Increases accuracy
double	8	Increases accuracy
long double	16	Increases accuracy

Table 5.2: Effects of modifiers and data type size

The way that a variable is as follows :

int variable_name;

long int Variable_Name;

Note that the C language is case sensitive, so in the above example two separated variables will be created. The size of the variables are dependent on the system that the programs is compiled on. Because of this a method of determining the size of a data type is required, especially for dynamic memory allocation. In C the function sizeof(variable) does exactly that, it's usage is as follows:

size = sizeof(variable);

This will return an integer variable which is the number of bytes that the data type occupies. By using this function then programs can be written to perform reliably no matter what the size of the data types, especially when the program is obtaining memory dynamically. An example program is shown in Appendix C, it shows how to define various data types and the use of the sizeof function.

The final data type that C provide is the pointer, this data type is used to store memory addresses. This data type allows C programs with a very powerful tool, as it allows for functions to access variable outside their scope and the allocation on dynamic variables. With the use of pointer the programmer is able to create abstract data types that can grow in size dynamically, which useful when the number of elements is unknown.

The basic way to define a pointer is by putting a * character directly in front of the variable name. The pointer can be of various data types so that the compile will know how many bytes there are to each variable. This is required for features such as the increment and decrement functions provided by C on pointers. The creation of a C pointer is demonstrated in the following code:

```
int *int_ptr;
int_ptr = malloc(5*sizeof(int));
```

The first line of code creates a pointer to a int type variable, at this stage the pointer is full of random garbage which happens to occupy the memory location assigned to it by the compiler. The second line user the malloc function to create a block of memory with 5 integer sized variables, note that the 5 could easily be a variable with any value. If the malloc function is successful in the creation of this memory block then the address of the first byte of the block is returned and entered into the int_ptr variable, however if malloc fails NULL is returned.

The most important thing to note here is that the variable int_ptr contains the address of the variable, hence to access the contents of the memory address the indirection operator (*) must be used. This must be placed before the variable name, similar to the way the pointer is created, then the pointer can be used just like a normal variable. This property will be shown in the following example:

```
*int_ptr = 4;
printf("The pointer contains %d \");
```

5.1.2 Data Arrays

The previous data types are only capable of holding one variable at a time. In most cases such as a string of characters are need, especially when writing programs for the GPIB as the commands to the devices are comprised of string of ASCII characters. So a way in which logically grouped data can be stored is required.

The C/C++ languages provide the user with the ability to define an array of a specific data type of any dimension. The way in which an array is created and indexed is as follows :

```
char array[10];
int intarray[10][10];
array[0]= 'h';
intarray[0][0] = 2;
```

With indexing an array in C the index range is from 0 to the size-1, so in the example above the index is 0 to 9. The second line in the example shows how a multi dimensional array is created, the indexing is the same as a single dimensional array except an index for each dimension is required.

5.1.3 Structures

In most cases it is common that different data types are required to identify an object, for example a student may have a Name, age, Student number and grade point average. So it makes sense to group this type of data into one entity, this is the purpose of C structures and C++ classes. The following code is an example of how a structure is created and defined as a new data type in C is as follows:

```
typedef Struct Student {
   int age,student_number;
   char name[30];
   float GPA;
} STUDENT;
```

After the structure is defined using the typedef function then the user can create a variable of this type in the same manner as a standard data structure. The label assigned to the data types in the string after the right parentheses and before the semi colon, in this case STUDENT. The creation of a student variable is as follows:

```
STUDENT St1;
```

After the variable has been created then each of it's elements can be access using the '.' operator. The following shows the student data type being used:

St1.age=18;
strcpy(St1,"Nathan");
printf("Age is %d \n",St1.age);

A full example program of the use of structures is found in Appendix C.2 .

5.2 Variable Scope

One of the most important considerations whenever writing C programs is the scope of a variable. The scope of a variable as defined by [16] is the range of statements in which the variable is visible. If the scope of variables is not considered then this can lead to hard to find bugs. A small sample program below displays variable scope:

```
/**********************************************************
* File Name: scope.c                                     *
*                                                        *
* Description: This program demonstrates the scope of    *
*              variables in C                            *
*                                                        *
* Written By: Nathan Hetherington                        *
**********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int x;

// Defines sub routine
void sub1(void){
        int x=50;
        printf("x is %d in Sub1\n",x);
}

int main(void){

        x=10;
        printf("x is %d in the main function \n",x);

        sub1();

        printf("x is %d after sub1\n",x);

        sleep(5000);
        return 0;
}


```

The results from running this program after it is compiled results in the following output :

```
x is 10 in the main function
x is 50 in Sub1
x is 10 after Sub1
```

From examining the output the variable x defined before the main function is seen through out the whole main function, the x variable defined in the Sub1 function is only visible to the statements in sub1. If Sub1 was meant to change the x variable, then this could lead to problems and in a program with more line could be difficult to find. This shows that create care with respects to variable scope must be taken when writing C programs.

5.3 Input/Output in C

The C programming language allows the user to read and write to what is referred to the stdin(Standard Input) and stdout(Standard Output). This standard input and output is by default the keyboard and monitor respectively, but C also allows the user to specify a file or some other I/O device. The interfacing with I/O is done via functions provided by the C standard libraries , such as stdio.h and stdlib.h, or API libraries for the I/O device such as the sicl.h and visa.h libraries for this project.

5.3.1 Standard I/O with keyboard and monitor

There are a few functions provided by the standard C libraries that read from the keyboard, their use depends on the type of data desired. Two such functions are *scanf* and *gets* , the first obtains a single variable of the type specified and the second is used to get a string of characters. The output is handle by the one function *printf* and prints the contents of the variables passed to it, the way the variable is interpreted depends on the variable format identifier used.The following is a table of some of the format identifiers available:

Format Identifier	**Variable type**
%d %i	Decimal signed integer.
%o	Octal integer.
%x %X	Hex integer.
%u	Unsigned integer.
%c	Character.
%s	String.
%f	double.
%e %E	double.
%g %G	double.
%p	pointer.
%n	Number of characters written by this printf.

Table 5.3: Format Identifiers used with I/O functions

The *printf* function can be called to print either a constant string that the programmer hard codes in, a variable within the program or a combination of the two. The code below shows an example of each :

```
printf("This is a Constant String\n");
printf(" %d %f \n",int_var,float_var);
printf(" The integer variable has the value %d \n",int_var);
```

From the examples above we can see that the printf function requires 2 arguments. The first is the string that will be printed, this string must contain formating identifiers and control characters (\n for new line). And the second argument is the list of variable that correspond to the format identifiers. Care must be taken to ensure that the variable names are in the right order, otherwise incorrect data will be printed.

The scanf function is as follows :

```
scanf(%d,&int_var);
```

This shows that the scanf function also requires two arguments, the first tell the function what type of data is expected and the second argument is the variable that the data will be stored in. The important thing to note is the use of the ampersand operator, this operator in C means "*the address of*", and is important for the function to know the memory address of the variable.

5.3.2 Using files for I/O

C allows the user to use files for input and output and does so in a similar manner to the standard I/O functions. The only difference is that an ID for the file is needed, and is created using the *fopen()* method and after used the file must be closed using the *fclose()*. These functions are used in the following manner:

*FILE *id;*
id = fopen("Filename.txt",'r')

In order to read and write to files a pointer to the file must be created, this pointer is type FILE and is defined in the stdlib.h C library. Once a pointer of this type is created then the *fopen* method is used to link the file to the pointer, and it requires two arguments. The first of these arguments are the file name that is in the form of a string, this string can be hard coded in of enter by the user into a character array. The second argument is the mode the file will be opened in, and there are a few different modes that can be used when accessing a file, they are as follows:

Format Identifier	**Variable type**	**File Created?**	**If File Exists?**
a	Appending .	Yes	Append to
a+	Reading and appending.	Yes	Append to
r	Read only.	No	If not found returns NULL
r+	Reading and writing.	No	If not found returns NULL
w	Write only.	Yes	Overwritten
w+	Reading and writing.	Yes	Overwritten

Table 5.4: *fopen* File modes

After the file has been linked to the pointer, then the file can be written to and read from. These task are performed using the *fprintf()* and *fscanf*, which are similar to the function for standard I/O except they require file pointer. The Examples below demonstrate this:

```
fprintf( ID, "Number %d:\t %d\n", i+1, score[i] );
fscanf( fp, "%d", &i);
```

From the examples above it can be seen that their use is exactly the same as to standard I/O, except for the addition of the file pointer. All format identifiers and control sequences are exactly the same.

5.4 Functions

Within C this allow the user to write functions that perform some task and can be called by the user in a similar manner to those defined by the C libraries. The advantage of this is that the user can write a function that performs a certain task, and call it when desired instead of replicating the code each time the task is needed. The following code defines a function that multiples two numbers, of course a function may do more complex thing then this :

```
int mult(int x,int y){
   int z=0;
   z = x*y;
   return z;
}
```

From this we can see the basic structure of the a C function. This first important value is the return type of the function, this tell what data type will be returned from the function. The next is the name of the function, and can be any name as long as it does not contain any of C's special characters. Finally the last element of a function is the arguments, this is the data that is passed to the function for processing. There is no real limits on the number of arguments that a function can have, but is up to the programmer to limit this to a reasonable amount.

The main function is a special case of a function, but the same rules exist. Finally if no return type or arguments are needed then the *void* data type can be used. And the user is referred to Appendix C for sample programs.

Chapter 6

Common Gateway Interface

The Common Gateway Interface as defined by [3] is a standard for interfacing external applications with information servers, such as HTTP or Web servers. This allows the user to pass information that can be processed by the application to achieve some goal, this type of interface is used for a large variety of tasks. These task range from simple applications that query a database or process order information for On-line shopping, to more complex applications that control equipment such a space telescope or programmable electronic test equipment, the reason for this project.

When using the Common Gateway Interface, the web server allows the user to run the application locally on the server. This is the most important point, the application is not run on the user's system, it is run on the system that the web server is installed on. So because of this fact the programmer must take some precautions writing CGI programs to ensure the security of the system is preserved. As outlined in [3] this security is usually controlled by the administrator of the web server, and is performed by controlling the directories that the user can access. The CGI applications reside in their own directory, this informs the web server to execute these file rather that display them on the browser.

To ensure a more efficient and compatible system, CGI programs can be written in a variety of programming languages. The most used languages for CGI programs are C/C++, Perl and Unix Stripting Languages (BASH,KORN, etc.). If a language such as C/C++ or Visual Basic are used, then the program must be compiled and the resulting executable placed in the cgi_bin directory of the web server. In the case of scripting languages such as Perl which are interrupted languages rather then compiled, the script can be placed directly into the cgi_bin, and an interpretor will need to be installed and functioning on the system. It is worth noting that the interpreted languages will be slower than their compiled counterparts, so if efficiency is of importance then a language such of

C/C++ should be used.

6.1 HTML CGI Program Calling Method

The common way that CGI programs are integrated into web pages, is to have a button that once click causes the application to be run. The HTML code that does this must also specify the method which will be used for passing the information to the application. The code below is responsible for this :

```
<FORM METHOD=”POST” ACTION=”/cgi-bin/cgipost.exe”>
... Code for data entry goes ...
<INPUT TYPE=”submit” VALUE=”Press to Submit”>
</FORM>
```

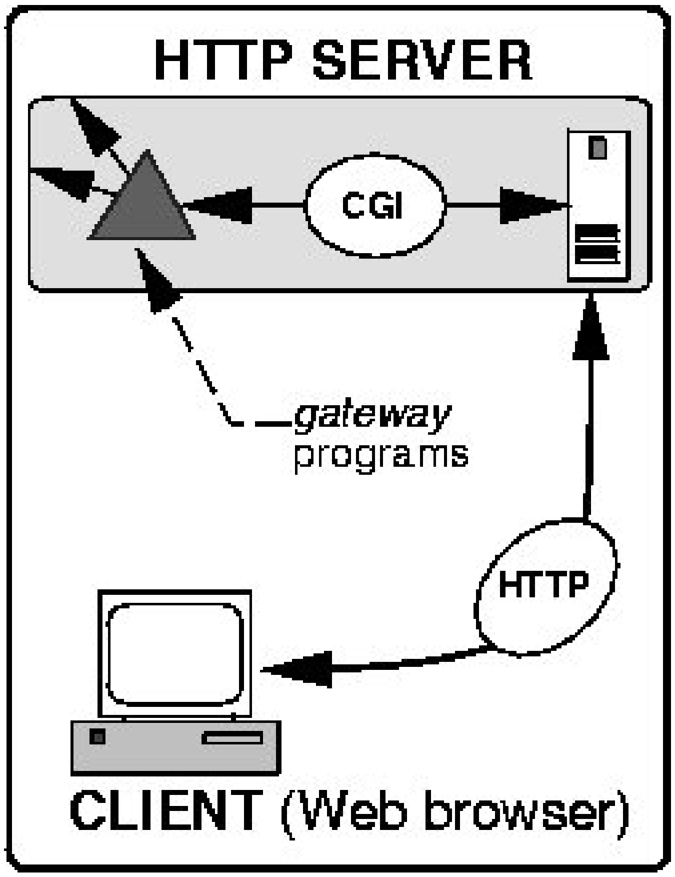
This input is pretty self explanatory, it consists of the method of input data and the application to run when the submit button is pressed. The string in the action section contains the path of the application, and is referenced from the web servers main directory. There are no limitations on the number of forms displayed on the one web page, so it is possible to have multiple submit buttons corresponding to separate applications. Caution must be taken when using multiple forms to ensure the correct application is called for each data set.

6.2 CGI Program Basic Structure

The basic operation of the CGI is shown in the following picture obtained form [4]:

From this figure, it can be see that is the Common Gateway interface is comprise of the three components mentioned in Chapter 4. This section will deal with the creation of the Gateway programs depicted in the diagram. All CGI applications will follow the same basic structure, and are comprised of three sections. These are obtaining input from the user, performing applications task and outputting a web page to user containing the status of the application. The reader is referred to Appendix E.1 which contains CGI example programs obtained from [13].

Figure 6.1: Flow of data to a gateway application



6.2.1 Obtaining Input

The main purpose of a CGI program is to obtain information from a web page, then use this information to carry out some tasks. There are two way that a CGI program can obtain data from a web page, and these are:

1. **GET** – the data are passed within the query string of the URL. For example, accessing the URL http://server.edu/stuff/cgi_program?query_string sends the data included in query_string to the HTTP server running on the machine server.edu.

2. **POST** – the data are sent as a message body that follows the request message sent by the client to the server. This is more complex than GET, but allows for more complex data. This method uses Standard I/O to pass the data to the application.

Looking at the examples in Appendix E.1 it can be seen that the get method uses environment variables to pass the data to the application. The application then reads the environment variables by using the *getenv()* function. This function requires the name of the environment variable to be obtained, and returns the contents of the contents of the environment variable specified by the argument.

The environment variable that is on interest in the get method is QUERY_STRING, this is a string that contains all the information from the web page that the application is to process. The format of this string is the same no matter weather the GET or POST method is used, and is as follows:

```
TEXT_LINE_ONE=some+text&TEXT_LINE_TWO=default+value
&TEXT_LINE_THREE=1234&PASSWORD_FIELD=secret
```

The sections in capital letters are the names of the input fields on the web page, the field name and value are separated by the '=' character and any white space is replaced by a '+' character. Each of the separate fields are separated using the & character, so the application must use these special characters to filter out the desired data.

The post method is much simpler to use, as the data is read from the Standard input. This is done using the following line :

```
bytesRead = read(STDIN, &readBuf[0], READ_BUFLEN);
```

The function requires three arguments, then first specifies what input to read from and in this case is STDIN, which is defined previously as 0. The second argument is where the data read from the standard in is to be stored, the text in this argument specifies the first byte of the buffer *readBuf.* The last argument tells the function the size of the buffer, this is so it can keep track of how much more data can be placed in the buffer.

6.2.2 Processing Data

As mentioned previously the data is sent to the application as one large string, this string must first be processed and the different elements separated. After the string has been processed and it's elements are obtained and stored in different variable, then the application can use the data to perform any task desired. In the case of this project after the data has been obtained then it can be used to change the settings on the Automated Test Equipment.

6.2.3 Output from CGI Application

After the application is finished performing all the tasks it is required to, then it must respond to the user with some feedback. This response must be in the from of a HTML web page and must contain all the headers and trailers that a standard web page contains. This can be seen in the examples in Appendix E.1.1, with lines 51 to 62 of cgiget.c defining the HTML header and 262 to 264 adding the HTML trailer.

Any code can be entered between the header and trailer as long as the code follows the HTML format. The programmer can use any functions from the string.h library to manipulate the strings that will be displayed on the web page, and all information will be in the form of ASCII characters. Because the program is what is responsible for the reply, then the information that is sent back can depend on the results of the processing, making the application capable of dynamic web development. The sending of data back to the user is performed using the *printf()* within C, which prints to the standard output.

Chapter 7

Writing CGI Programs for Devices

The programs that are written for interacting with the devices of the ATE, will follow the structure outlined in the previous chapter on Common Gateway Interface. The cgi program written for the Function generator will be the eample referred to in this chapter while describing the process involved in the creation of these programs. The process described for this application is exactly the same for each device, with little variation, so can be used to create programs for the different programmable instruments available.

7.1 Step 1: Open a Session with the Device

In order to communicate with a device, the programmer must create an interface session with the device. The method to do this was outlined in chapter 3, where the *iopen()* function is used to open a session and requires the address of the device. In the CGI program for the 33120A device, the address can be defined as a constant using the #define mechanism in C.

```
#define DEVICE_ADDRESS ”hpib7,10”
```

This allows the programmer to use DEVICE_ADDRESS whenever the actual device address is required, during compilation this is replaced with ”hpib7,10”. This is not compulsory, but makes the code easier to understand and change later. If the address of the device changes then only the definition string will need to be updated. Using this address, a session can then be opened and the resulting ID will be use when communicating with the device. At the completion of the program this session must be closed using the *iclose* function, failing to do so will cause later run programs to act unpredictably.

Device	Address
33120A Function Generator	10
54602B Oscilloscope	7
34970A Switch Unit	9
6624A Power Supply	5

Table 7.1: Addresses of Devices in ATE Rack

7.2 Step 2: Getting Data from Input String

As mention previously the data that is enter into the fields of the web page are read into the program as one string. So the application must separate the data from this large string. The format convention of the input string is, a = is used to separate the data label and actual data and the & is used to separate the different data inputs.

So the programmer can simply step through the input string, looking for the '=' character. Once this character is located then all characters between it and the next & character is the data wanted. In the CGI programs there is a function that takes the input string as the input argument and steps through the array of characters, separating the various data. The user is referred to the CGI program for the function generator, the code for the function *getVariables(char *Buffer)*. This function will change from application to application, and it's sequencing is dictated by the sequence on the data fields on the corresponding web page.

7.3 Step 3: Obtain Device Commands

The first step in the creation of the device command is to obtain the syntax of the commands that will be sent to the device. The best source for such information is the User's Manual or Programmers Manual produced by the manufactures of the instrument. For this project, these documents where obtained from the Agilent Technologies web site.

Once the device settings being changed have been determined, then the syntax of the command to be sent to the device must be looked up in the manual. All commands are hierarchal, this means that commands have subcommands that can be used in conjunction, to specify the exact setting of the device to be changed. One example of this is the APPLy command, that allows setting to be applied on the device, some of the sub-commands are:

```
: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>] ]]
```

From the above examples, the different are fields have the following meanings:

- Square brackets ([ ]) indicate optional keywords or parameters.

- Braces ({ }) enclose parameters within a command string. Default parameters are shown in bold.

- Triangle brackets (< >) indicate that you must substitute a value for the enclosed parameter.

So using the information contained above, a string can be generated to change setting can be generated. This string can then be sent to the device, and the desired changes will be made. For example the following code will set the function generator to produce a sine wave with a frequency of 12Khz, 2V peak to Peak and an offset of 0V.

```
APPL:SIN 12Khz 2V 0V
```

Note that a semi-colon is used to separate the different command levels and there is a space between the different input fields. When this string is sent to the device, a newline character must also be sent to notify the device of the end of the command. In the program written for function generator, the SIN section of this command string depends on what is selected by the user.

7.4 Step 4: Changing Settings on the Device

Using the command created using the data obtained from the user and the commands for the device, demonstrated in the previous step. The string can then be sent to the device using the *printf()* function. This is done in the following way:

```
iprintf(id,"APPL:SIN %s,%s, %s\n",Frequency,PtPVolts,Offset);
```

This is the line in the code that will actually implements the changes on the device.

7.5 Step 5: Error Detection

After the string has been sent to the device, then it must be checked that the change was actually implemented. The easiest way to check that the changes have been implemented is to query the device for it's current setting, this can then be compared to the string that was sent to the device. The devices current settings can be obtained by using the *iprompt()* function.

```
ipromptf(id, "APPL?\n", "%t", temp);
```

This will cause the current setting to be in the variable temp as a string, this can then be compared using the string compare function *strcmp()*. If the two strings are the same then strcpy will return 0, and then the appropriate action can be taken.

7.6 Step 6: Output Results

Depending on the result of the strcmp function, a result must be sent to the user to inform on the status of their request. If the settings of the device where successfully changed, then a web page stating that the settings have been applied will result. If the program fails, then the user is informed to try again. There are nothing that the user can do if the problem is with the hardware, in this case an error file is written and it is up the administrator to check this file. Later implementations may sent an e-mail to the lab technician, informing them that an error has occurred.

7.7 Compiling Device Programs

When compiling programs that make use of the SICL library, then some sets must be taken. First is the inclusion of the sicl.h header file, which contains the prototype definitions of the functions provided by the library. And finally, is informing the link which library file contains the machine code for the functions defined in the header. In the case of this project sicl32.lib was the appropriate file.

Chapter 8

Creating Web Pages for Remote Access

The Hyper Text Mark-Up Language is a plain text based language, and can be created, edited or viewed using a simple text editor on any computer platform. HTML is similar to a scripting language as the web browsers reads the HTML file and determines the way of formating the data from the code. This makes HTML multi-platform as long as a web browser exist for that platform. Also with the modular structure of the web browser, this allows for different services to be provided via plug-in. This makes the power of HTML almost unlimited.

8.1 HTML Basics

The HTML language is based on what is known as HTML tags, these are used to notify the browser of formatting method. The basic structure for all HTML tags are <HTML_TAG> to indicate the beginning of an HTML environment and </HTML_TAG> to finish the effect of this environment.

The first tag that must exist is the <HTML> tag as it tells the web browser where the HTML code begins and ends. This alone would created a valid web page and could be loaded, but the result would be a blank screen. After the beginning and end of the HTML file is defined there are two more sections that need to be created.

The first of these sections is the HEAD, this section contains information about the web page. One such element is the TITLE of the web page, the contents of this field will appear in the browsers title bar. Additionally the contents of the

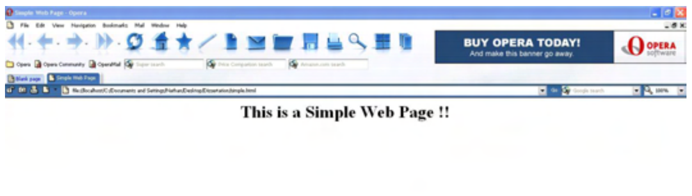
TITLE field will be used for the history list of the browser and also will be the title entered when the page is book marked.The final section that a web page will consist of is the BODY, this section contains all the information that is to be displayed on the web page.

The following code will produce a simple template for a web page:

```
<HTML>
<HEAD>
<CENTER><TITLE> Simple Web Page </TITLE></CENTER>
</HEAD>
<BODY>
<H1>This is a Simple Web Page !!</H1>
</BODY>
</HTML>
```

And the resulting web page from the code above is :

Figure 8.1: Simple HTML page



This is a Simple Web Page !!

From the simple template, this code can be added to to make more complex HTML pages. There are heaps of HTML tags that allow the developer to customize the appearance of the web page they are developing. Typing text into the body section of the HTML page will result in the text being displayed in the default font and size. But tags must be used to format the appearance of the text,tables, division lines and other elements.

The HTML language provides the developer with a large range of features, just like a word processor, to create format documents. The developer can insert pictures,table, itemized lists and change the appearance of various elements of their page. There are such a large number of element that can be changed through the use of HTML tags, that readers are referred to introductory HTML books such as [1] or the Internet for various formatting information.

Another important feature that HTML provide is the ability to create data entry field, these are what will be used by this project to pass data to the CGI programs. There a few different types of data entry fields, but the ones of interest for this project are, the text entry field and radio buttons. The first element creates a box in which plain text can enter, the maximum size of which can be set. The next has a grouped set of options, of which only one can be selected at any time.

The tags just mentioned will be displayed by the following code and explained after the code:

```
<!-- This creates a comment >
<HTML>
<HEAD>
<TITLE> Customized Web Page </TITLE>
</HEAD>
<BODY BACKGROUND="background.jpg" TEXT="#FFFFFF">
<CENTER><H1>This is a Simple Web Page !!</H1></CENTER>

<P>This is an Itemized List:</P>
<HR>
<UL>
<LI>Item 1</LI>
<LI>Item 2<UL>
   <LI>Item 2A</LI>
   <LI>Item 2B</LI>
 </UL></LI>
<LI>Item 3</LI>
<LI>Item 4</LI>
</UL>
<HR>

<P>Radio Buttons:</P>
<input type="radio" name="group1" value="Button1" checked> Button 1<br>
<input type="radio" name="group1" value="Button2"> Button 2<br>
<input type="radio" name="group1" value="Button3"> Button 3<br>
<input type="radio" name="group1" value="Button4"> Button 4<br>

<HR>
```

```
<TABLE align="center" border="3" width="668" cellpadding="3" id="table1">   29
<TR>                                                                         30
  <TD width="222">Text 1 :                                                   31
  <input type="text" name="text1" size="20">                                 32
  </TD>                                                                      33
  <TD width="222">Text 2 :                                                   34
  <input type="text" name="text2" size="20">                                 35
  </TD>                                                                      36
</TR>                                                                        37
</TABLE>                                                                     38
                                                                             39
</BODY>                                                                      40
</HTML>                                                                      41
```

This code contain only a small proportion of some of the features provided by the HTML language, but the use of options shown in the example demonstrates the convention used for HTML options. Line 6 shows how options for the body section can be used to alter the appearance of the overall page, the BACKGROUND option allow the background of the page to be set to a color or an image, the TEXT option defines the fonts color.

The use of the HR, P and BR tags show how they can be used to alter the appearance of the page. The P tag defines a paragraph, and creates the spacing before and after the text that the code encloses. The BR tag creates a line break and does not require a </BR> tag , this is the same as <HR> except a horizontal line is created across the page.

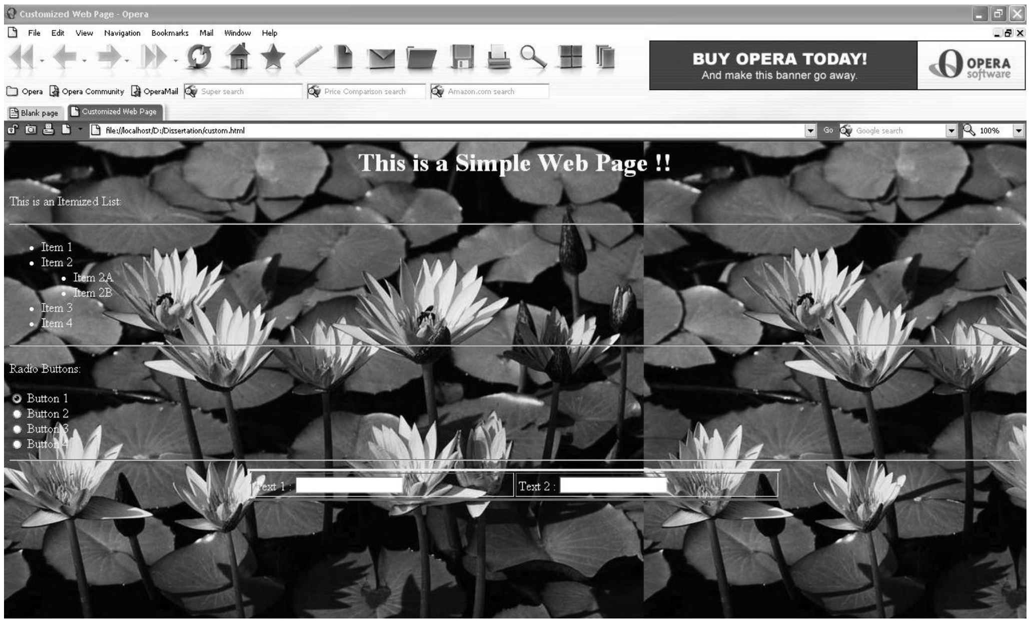
HTML provides the ability to create a list, the list will be numbered or have bullet points depending what tag encloses the list. In this case the UL tag is used and produces a bullet pointed list, if OL was used then the list would be numbered. Each element of the list enclosed within the list type are specified by the LI tag. The addition of another list type can be used to create a sub list within another list.

The radio buttons can contain an unlimited amount of elements, but only one element can be selected at any time. The group is specified using the name field, all elements with the same name will belong to a group. The value field is used to separate each of the elements of a group, this is so that the individual button can be identified. One final feature of the radio button is the ability to specify a default value, this is so that one of the group is always selected, and is done using the checked option.

Finally is the text entry field, this element allows the user to under strings that can be used by a CGI program. This element is created using the input tag, the type of input is specified using the type field and the size option specifies the number of characters that can fit in text field. Just like the radio buttons the text field has it's own unique name, specified in the name field. The way that the text fields are displayed in the above example, uses a table structure which allows the creation of a matrix with the text fields as the elements. This structure is simple to create, with the ¡TR¿ tag used to create a matrix row and the TD tags create columns. This allow the developer to group similar fields to create some logical grouping of data.

The figure below shows the page resulting from the code :

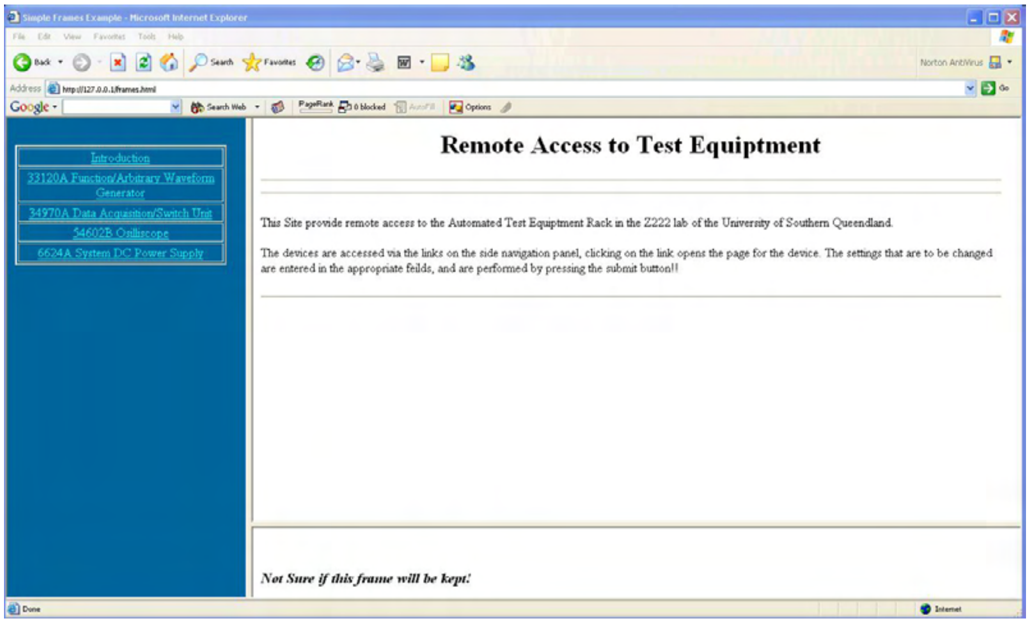
Figure 8.2: Web page with Background and other elements.


Customized Web Page - Opera
File Edit View Navigation Bookmarks Mail Window Help
BUY OPERA TODAY!
And make this banner go away.
OPERA software
file://localhost/D:/Dissertation/custom.html
This is a Simple Web Page !!
This is an Itemized List:
Item 1
Item 2
Item 2A
Item 2B
Item 3
Item 4
Radio Buttons:
Button 1
Button 2
Button 3
Button 4
Text 1 :
Text 2 :

8.2 Frames

Also HTML allows the developer to create what are called frames, which allows multiple Web Pages to be displayed at one time. Using this structure the developer can use one of the HTML pages as a navigator to the other pages. This is the method used to provide access the the different programmable instruments in the ATE rack. The following figure displays the main page designed for this project, and the code after the figure shows how a form is created:

Figure 8.3: HTML Frames


Simple Frames Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Norton AntiVirus
Address http://127.0.0.1/frames.html
Go
Google Search Web PageRank 0 blocked AutoFill Options
Done
Internet

Introduction
33120A Function/Arbitrary Waveform Generator
34970A Data Acquisition/Switch Unit
54602B Osilliscope
6624A System DC Power Supply

Remote Access to Test Equiptment

This Site provide remote access to the Automated Test Equiptment Rack in the Z222 lab of the University of Southern Queendland.

The devices are accessed via the links on the side navigation panel, clicking on the link opens the page for the device. The settings that are to be changed are entered in the appropriate feilds, and are performed by pressing the submit button!!

Not Sure if this frame will be kept!

```
<html>

  <head>
  <!-- DOCTYPE HTML -->
  <!-- A Basic HTML Frames example -->
  <!-- Nathan Hetherington 2004 -->

  <title>ATE Remote Access</title>

  <!-- this is for use by web search robots -->
  <!-- "keywords" is used for searching -->
  <!-- "description" is the summary returned -->

  <meta name="author" content="Nathan Hetheington">
  <meta name="keywords" content="html">
  <meta name="description" content="Provide remote access to ATE equiptment">        16
                                                                                   17
  <!-------------------------------------------------------------------------------> 18
  <!-- 150 = pixels, * = remainder --><!-- *** warning: if only a single panel *** -->  19
  <!-- do not use nested frameset environment -->                                  20
  <!-- 10%=percentage -->                                                          21
  <!-------------------------------------------------------------------------------> 22
  <!-------------------------------------------------------------------------------> 23
  <!-------------------------------------------------------------------------------> 24
                                                                                   25
  </head>                                                                          26
                                                                                   27
  <frameset cols="300,*" border="5">                                               28
  <frame src="Frames_files/NavPanel.html" name="Nav">                              29
  <frameset rows="85%,15%">                                                        30
  <frame src="Frames_files/Project.html" name="Upper">                             31
  <frame src="Frames_files/Static.html" name="Lower" scrolling="no">               32
  </frameset></frameset>                                                           33
                                                                                   34
  <noframes>                                                                       35
    <BODY>                                                                         36
      <P>         This document uses frames          </p>                          37
                This is in the noframes section                                    38
    </body>                                                                        39
                                                                                   40
  </noframes>                                                                      41
                                                                                   42
</html>                                                                            43
```

The code for this page was obtained from [13] and altered to suit need of this desired implementation. The important section of this code is lines 28-33, as this section is responsible for the division of the page into frames. The *frameset* tag is used to specify the sections containing the information about how the page is to divided, this division can be vertically using the cols option or horizontally with the rows option. The size of the divisions are specified with the option used, and can be expressed as number of pixels or as a percentage.

After the frame sizes have been specified, the pages that must be loaded into each section must indicated. This is the purpose of lines 29,31 and 32 and various options can be applied to each frame, such as the no scrolling option that is enabled for the static frame. The division of the page and manner in which the pages operate is entirely up to the developer, and allows for the implementation of more powerful, user friendly sites.

8.3 Creating HTML pages for ATE Instruments

The creation of HTML pages is a creative process, and aspects such as appearance and operation can vary tremendously. But in the case of web pages for use with programmable instruments, there are a few aspects of the implementation that remain constant. For example the way that settings are changed is dependent on the way feature being set operates, and usually varies from device to device. Some setting of a device may be limited to a set of values, for example the wave type produced by the function generator, so for this reason radio buttons more suit the acquisition of such information.

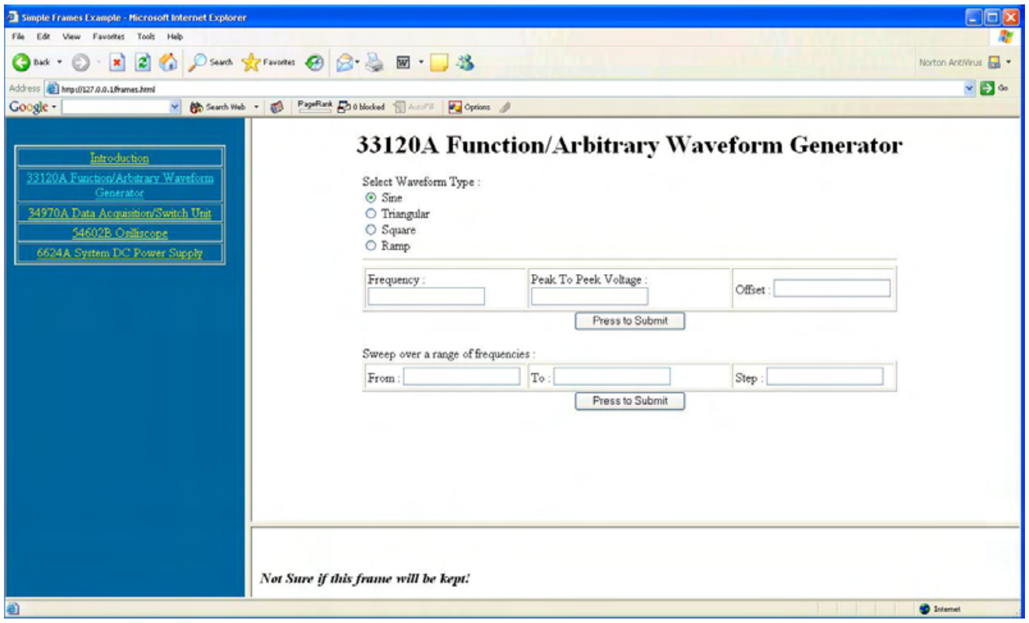
It is not totally necessary that the right input type be used, but it makes the development of CGI code easier. If a text box entry was used to acquire the wave type desired, then the CGI application would have to check that the entered wave type is valid and take action if it was not. With the use of the best suited input method, then the need to code for error detection and correction is not necessary.

So for the implementation of web pages for the devices, a simple analysis of the feature that the device provides was required. Features that only contain a discrete number of setting are more suited for implementation using radio buttons, if a continuous number of values where available then a text box should be used. The web page created for interaction with the HP 33120A Function Generator is shown in the following figure:

The function generator's web page above shows how multiple functionality can be implemented on the one page. For this web page the selection of the waveform type is done using radio buttons, this makes sense as there are only the four different waveforms that the web page will provide. The selection of frequency, peak to peak voltage and voltage offset is done using text boxes.

Creating web pages for each device allows the user to access and alter the settings of the specific device that the page was written for. But the desire to provide some form of sequencing to the user may be desired. This could be done by creating web pages for each step in the sequence, then via a navigation frame such as the one used for the main implementation above could be used to step through the steps. Providing

Figure 8.4: HP 33120A Function Generator Web Page


Simple Frames Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Norton AntiVirus
Address http://127.0.0.1/frames.html
Go
Google
Search Web
PageRank 0 blocked AutoFill Options
Introduction
33120A Function/Arbitrary Waveform Generator
34970A Data Acquisition/Switch Unit
54602B Osilliscope
6624A System DC Power Supply
Press to Submit
Press to Submit
Internet

33120A Function/Arbitrary Waveform Generator

Select Waveform Type :
[✓ Sine]
Triangular
Square
Ramp

Frequency :	Peak To Peek Voltage :	Offset :

Sweep over a range of frequencies :

From :	To :	Step :

Not Sure if this frame will be kept!

the sequencing in this manner would reduce the size an complexity of the CGI programs that are responsible for each step. This structure would also make it possible to use the individual components in a various number of sequences, and each step would be easily maintainable.

Alternatively the implementation of sequencing could be done via the use of a page with the individual steps all listed, with the submit button only available for the step currently available. Through the use of a complex CGI program for each step, a web page with the submit button for the next step being returned after the changes have been implemented. This manner of implementing the sequencing has the advantage of stepping the user through the step automatically, the main disadvantage of this method is the cgi cannot be used for another sequence. If a new sequence was desired then a new web page and corresponding CGI programs would need to be created.

Another aspect of creating web pages for the ATE devices, is the type and number of settings that should be implemented on each page. For some devices, such as the Power Supply, using one page to contain all the possible settings is sufficient. But in the case of other devices such as the Function Generator, Oscilloscope and MUX, there are far too many options to be included on the one page. So some form dividing the these setting must be decided.

As mention in chapter 7, the device settings are separated into sub command and using these divisions would make sense. This type of devision would make the development of CGI programs that are responsible only for a certain sub-command, decreasing their size and complexity. Also be cause the sub-division of command is done based on the function they provide, this type of division is logical.

But there is no need to implement all the commands and sub commands, as a majority of these features many not be need or could be out of the scope of the users knowledge. So providing access to only a small group of commands is also possible, and the commands omitted could be left as their default or set to a desire value. Creating the web pages in this manner controls the type of settings that the user can use, and helps in the coding of corresponding CGI programs. The code for the pages created for this project are located in Appendix D, these page contain only basic features for each device. The other features for the devices can be implemented in a similar manner.

Chapter 9

Setting Up an Apache Web Server

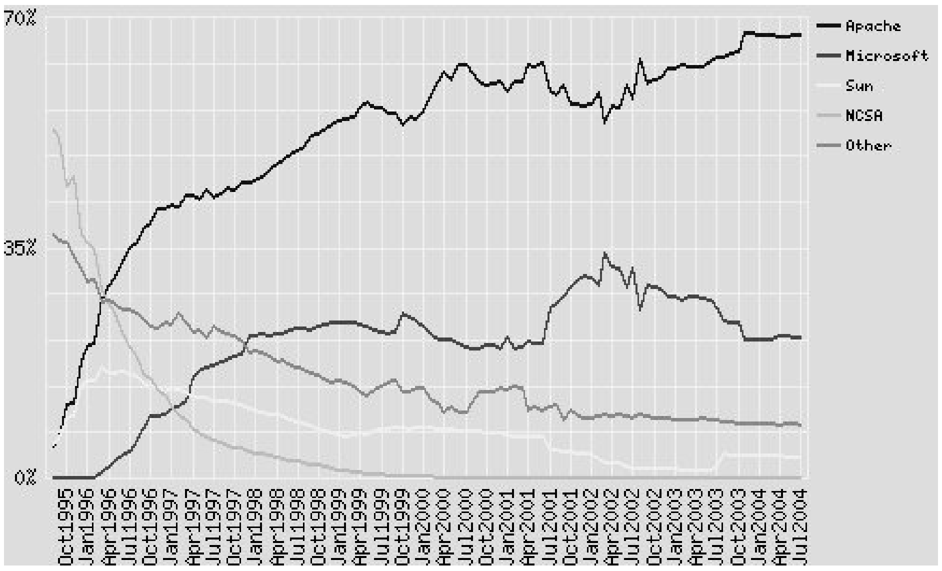
The web server used in the implementation of this system does not matter, as long as it is capable of providing the Common Gateway Interface method. There are various reasons for choosing Apache as a web server, for both this project and web hosting in general. This first is Apache's dominance in the web server domain, this is illustrated by the graph shown in the figure below. The only other web server that has a noticeable following is Microsoft's Internet Information Server, which is shipped with Windows Server operating systems.

In addition to it dominance of the market, Apache is also a cross-platform web server. This means that versions of Apache can be obtained for various operating systems, such as Microsoft Windows, Mac Os and Linux/Unix. From the view of this project this is beneficial, as the system could be implemented in another Operating System, inheriting their security and reliability advantages.

Probably the most appealing aspect of the Apache web server, is the amount of On-Line documentation and support that is available. This is due to the fact that Apache is an Open Source program, making it free to the public. And because this software is open source, this means that any software bugs and security holes are often detected and corrected much quicker then commercial products. Apache can be obtained from **www.apache.org** and is free for download in various versions and platforms. This site also contains a large amount of documentation for the installation and configuration of the apache server.

For this project an older version of apache web server was required, this is due to the fact that the system that apache was to be installed to contains the Windows 95 Operating System. Attempts to install new versions resulted in problems, mainly due to

Figure 9.1: Market Share of Various Web Servers



the fact that later versions are designed to take advantage of Thread technology. The Windows 95 OS is not a true multi-tasking OS and does not support the use of threads. Also it is worth noting that using Windows 95 for a web server is advised against, because of the security risks that the OS creates. So for further development of the process developed by this project, it is advised that a different OS is used for the system.

9.1 Installing Apache

As mentioned earlier, Apache can be downloaded from **www.apache.org** or various mirror sites and can be obtained in binary or source form. If the source code version is acquired, then it must be compiled prior to installation. The advantage of the source version is that in the compilation process, it can be optimized for optimal performance on that system. But a majority of the time the binary version is sufficient, and comes in .exe or .msi format.

There is not much difference between these two formats, except the msi files are smaller. The msi windows installer technology contains all data and instructions required to install the application on a system. In order to use a msi file then the msi.dll dynamic library must be installed, this is not necessary in later versions of Windows such as XP,2000.

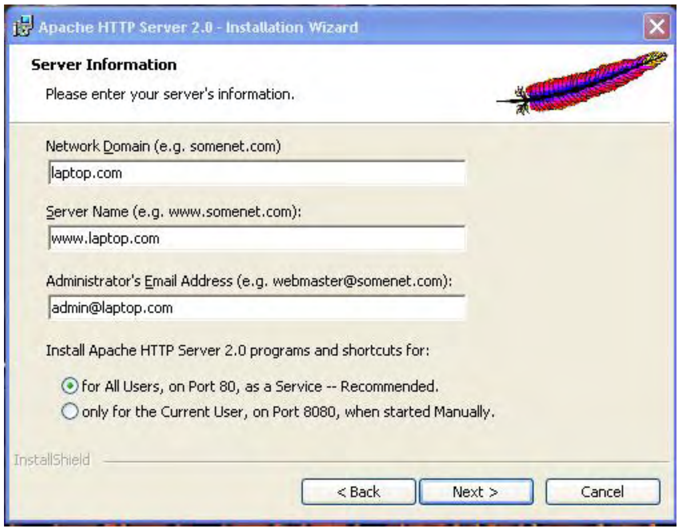
For the Windows 95 operating system, the winsock libraries have to be updated to winsock 2 before installing the apache server. The winsock libraries are what the windows OS uses for network communication, and contains functions for the creation, destruction and use of TCP communications. Web servers use TCP for their communications, which as stated in [18]:

1. Accept a TCP connection from client(a browser).
2. Get the name of the file requested.
3. Get the file (from disk).
4. Return the file to the client.
5. Release the TCP connection.

The file required to add support for msi files and update the winsock libraries are on the attached CD, or can be obtained from Microsoft's web site.

Once the desired version of the Apache web server is acquired, then simply executing the exe or msi will initiated the installation process. The installation process is pretty much self automated, except where information about the Domain and Administrators e-mail address are required. An example of this is shown in the figure 10.

Figure 9.2: Apache Installation



Apache HTTP Server 2.0 - Installation Wizard

Server Information

Please enter your server's information.

Network Domain (e.g. somenet.com): laptop.com

Server Name (e.g. www.somenet.com): www.laptop.com

Administrator's Email Address (e.g. webmaster@somenet.com): admin@laptop.com

Install Apache HTTP Server 2.0 programs and shortcuts for:

[✓ for All Users, on Port 80, as a Service -- Recommended.]

only for the Current User, on Port 8080, when started Manually.

9.2 General Configuration

Most of the configuration settings that relate to the operation of the web server are done via the httpd configuration file. This file is located in the confs directory, which is in the directory that the server was installed to. This file is a text based file, that is read by the server during loading.

An example of some of the configuration changes that are handled by this file, is the location of the web pages and the cgi programs. This allows the user to specify any directory that contains web pages that will be accessible for users. This configuration file is divided into three sections :

1. Global Environment
2. 'Main' server configuration
3. Virtual Hosts

The Global Environment allows settings such as the root directory where log file and configuration files are located. Also certain aspects of the operation of the web server can be changed via global environments, such as Timeout, Whether or not to allow persistent connections and maximum number of requests a server process serves. Aspects such as the number of request processed and the timeout are important for this project.

For the ATE remote access we want only one person changing the device settings at a time, using this global variable is not a very effective method of providing mutual exclusion. So for future implementation a better method of limiting access to one user at a time would be needed. Also the timeout property of the server may need to be changed, if the application applying the changes on the device takes to long the a time out may result. This could mean that a timeout may result when there is no problems on the server side, so increasing this timeout would help prevent this problem. But in order to not have the user waiting too long, so an approximation of the longest command processing time and a maximum round triptime for the request would need to be estimated.

The next section of the configuration file deals with the location of various files need by the server, including the HTML page that the server will provide access to. Along with this it also specifies the that access will be controlled, allowing or denying based on user names and passwords or IP addresses. This is where security settings can be enforced, through the selection of appropriate options.

Finally this last section allows the establishment of virtual hosts, which allows multiple domain/hostnames to exist on the one server. This maybe desirable if the server is to host web pages for different sites, such as a web hosting service would provide. In the case of this project there is no need to have multiple domains or hostnames, so this section will be left as is.

As mentioned earlier, there are many sources of apache web server configuration on the Internet. Hence the user is advised to search on the Internet for help regarding to the configuration of Apache Web Servers, if more detail is required. The site from which the Apache server is obtained, **www.apache.org** is recommended as a good starting point for such information.

9.3 Configuring Server

After the web server is installed, then all configurations are done via the httpd configuration file. This file is accessed and edited using a simple text editor, and contains code that look similar to HTML code. This file is read by the server application upon starting and specifies the operation of certain aspects of the server.

9.3.1 Controlling Access

For this project, there is the desire to provide a form of controlling access to this web site. Through the use of the httpd configuration file ,files created by the administrator and a file created by an application provide by the apache server, this facility can be provided.

The first stage in configuring controlled access is the creation of a password file, this is done using a program that comes with the apache server called *htpasswd*. This application is executed with three arguments, the first is the options that the user wishes to enable, secondly is the file that will store the user/password information created and last is the user that will be entered into the file. The basic operation of this application is as follows:

```
htpasswd -c /usr/local/apache/passwd/passwords username
```

The user will be prompted to enter the password, this will cause the creation of a file in the specified directory. This file contains the users name and the encrypted version of the password, this is so later authentication can be achieved by comparing the encrypted version of the enter password with that in the file. Note that the -c flag is only use to create the file, if a user is to be added to an existing password file then this flag is submitted. This is very important as using the -c flag will cause the file to be overwritten.

Care must be taken in the placement of this file, as it contains password information that could be used to guess the actual password. For this reason this should be placed in a directory that the http server cannot access. Also the permissions of the file must be set to allow access by the web server user, hence deny access to other users.

After the password in created and placed in a directory, then the configuration files must be set to use this newly created password file. This is done by creating and placing a file in the directory that is password protected, this file must be give the name htaccess. The fields that must be entered into this file are now describe :

AuthType	Authentication type being used.
AuthName	The authentication realm or name.
AuthUserFile	The location of the password file.
AuthGroupFile	The location of the group file, if any.
Require	The requirement(s) which must be satisfied in order to grant admission.

The htaccess file created for the project's web server is as follows:

```
AuthName "ATE User Login : "
AuthType Basic
AuthUserFile "c:/Program Files/Apache Group/Apache2/conf/htpasswd"
AuthGroupFile /dev/null
require valid-user
```

From this it can be seen that the basic authentication is used to control the access to the devices, this could be and of the various authentication setting depending on the desired effect. The path to the password file is specified by the AuthUserFile line, and was placed in the conf directory which is only accessible by the web server. And the required field specifies that in order to be granted access the user must be have an entry in the password file, and the passwords must exist.

Finally after all files have been created and placed in their correct directories, then the httpd configuration file must be edited to inform the server to use password access. This is done by editing *AllowOverride* option of the <Directory ”C:/Program Files/Apache Group/Apache2/htdocs”> section, which specifies access to the web pages in this directory. The AuthConfig option is used to activate the password access, after all these steps are done then the web server must be restarted.

9.4 Adding Web Pages and CGI Programs

After the web server has been installed and configured as desired, then adding the contents of the web page is simply a case of placing the files in the correct directory. In the case of the html pages and cgi programs, this is the directory specified using the <Directory ... >.

The defaults for these fields are the htdocs and cgi-bin directories located in the directory where the web server was installed on the system. Once the files have been placed in their correct directories, then any links or references to these files in the HTML pages must be updated to reflect there current placement.

Chapter 10

System Test

The testing of this system was simply a matter of seeing weather it worked under certain circumstances, as there are no real other measurable performance aspects. The system must be test in a manner that would simulate it's normal operation if implemented in the real world. If operation under these circumstances then there is no reason why the system would not operate correctly through the Internet.

The ideal way of testing this system, would be to provide the remote access system to be accessible via the Internet. But this was not possible, due to difficulties imposed by the information technology services due the possible security faults. This is understandable as the windows 95 has many security faults, and is not recommended as a web server as specified by the apache web server developers.

So in order to test this system and eliminate any chances of compromising the security of the Universities network, the different components that a majority of the Internet used. The first element is over an ordinary Ethernet network, once success has been proven on this configuration, a router will be implemented. The router will allow the testing of how the system operates, when going from one network to another.

For the test of the system over an Ethernet, the PC's where given IP addresses 192.168.0.1 and 192.168.0.2 respectively.The second address was assigned to the PC running the remote access system created by this project. Gaining access to the system from the second PC, was simple a matter of entering 192.168.0.2\frames.html into the address bar of the web browser of the first PC.

Once performed the introduction web page loaded, and the function generators page was chosen. Entering data into the fields and pressing the submit button resulted in the desired changes on the instrument, then another device was chosen and also proved successful. This means that the system operated successfully as predicted, under the first test circumstances.

The next test required the configuration of a router, in this case a laptop with an on-board network card and a PCMCIA network card. One of the Network Interface Card(NIC) was given an address of 192.168.0.1 and the other 192.168.2.1, and an entry was entered in the routing table of this PC so that information from one network to the other was passed to the correct interface device. Two PC's where connected to the laptop through the use of two network hubs, and each PC assigned an IP address that was from the same network of the NIC device of the laptop connected to the same hub. The system was tested using the ping command, and it was found that each host was reachable through the router.

Now the same test as previously used was duplicated, except the IP address was 192.168.0.2, and the PC being used was on a different network. The system was also proven successful under these circumstances. There was no noticeable delay, due to the router, but this was expected as on the Internet usually more than one router exists and the networks are processing more of a load.

So from these test, it is expected that the system will operate correctly over the Internet. Of course a much longer delay would be expected, but not enough to effect the system's performance. This is due to the fact that more routers would be traversed and the propagation delays caused by the communication media, the usual round trip time is around 200 ms. This would be noticeable, but the system only sends a small amount of data and would not be expect to cause the system to fail.

Chapter 11

Conclusion

At the completion of the thesis the system provides a small amount of functionality for each device, and web pages through which the interactions are performed. An Apache web server that hosted these web pages and provided password controlled access, was installed and configured. This was the major achievements of the project.

The web pages created to provide remote access to the devices could be used to provide access to the ATE Rack for external students. This would benefit student that would normally have to travel long distances to access these devices. Also assignments and practical assessments could be created using web pages and CGI programs, for students to complete remotely.

Also it is suggested that the systems that the ATE equipment is attached to is updated with more powerful computers. This will improve the security and performance of both the existing system and the system produced by this project. It will also help improve the development and testing of software developed using the methods outlined in the thesis.

The initial direction of this project was not as the author expect, with a client/server software development. The development of a web based system was emphasized by the Supervisor, as it solves some of the existing problems, so this was fully developed.

11.1 Suggested Further Development

Improvements to the current remote access system could include:

- The implementation of more functionality for each device.

- Development of a sequence of events for some test.
- Possibly the implementation of a graphical interface using Flash media web pages.
- Implement CGI programs using the VISA API library.

Bibliography

[1] Michael Anderson. *HTML Complete*. Sybex, San Francisco, 1st edition, 1999.

[2] UNKNOWN AUTHOR. *Danger of Open Port 139*. ONLINE, http://expertanswercenter.techtarget.com/eac/knowledgebaseAnswer/0,295199,sid63_gci980344,00.html, ACCESSED 14/10/2004.

[3] UNKNOWN AUTHOR. *The Common Gateway Interface*. ONLINE, http://hoohoo.ncsa.uiuc.edu/cgi/, ACCESSED 17/8/2004.

[4] UNKNOWN AUTHOR. *An Introduction to The Common Gateway Interface*. ONLINE, http://www.utoronto.ca/webdocs/CGI/cgi1.html, ACCESSED 18/8/2004.

[5] Paul G. Green. *Intelligent HPIB Intrument Control*. USQ, Toowoomba, QLD, Australia, 1st edition, 1992.

[6] Ron House. *Beginning with C*. THOMAS NELSON AUST, 2nd edition, 1994.

[7] Michael I. Hyman and Robert Arnson. *Visual C++ 5 for Dummies*. IDG Books Worldwide, Inc., 919 E. Hillsdale Blvd, 2nd edition, 1997.

[8] Agilent Technologies Inc. *Agilent 33120A Function/Arbitrary Waveform Generator - Quick Reference Guide*, 1997-2003.

[9] Agilent Technologies Inc. *Agilent 33120A Function/Arbitrary Waveform Generator - User's Guide*, 1997-2003.

[10] Agilent Technologies Inc. *Agilent 34970A Data Acquistion/Switch Unit - Quick Reference Guide*, 1997-2003.

[11] Agilent Technologies Inc. *Agilent 34970A Data Acquistion/Switch Unit - User's Guide*, 1997-2003.

[12] Richard Johnsonbaugh & Martin Kalin. *Object-Oriented Programming in C++*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 2000.

[13] John Leis. *HTML,Forms and CGI.* ONLINE, http://www.usq.edu.au/users/leis/notes/html/index.html, ACCESSED 4/6/2004.

[14] Hewlett Packard. *HP Standard Instrument Control Library - User's Guide for Windows*, 3rd edition, 1996.

[15] John M. Pieper. *Automatic Measurement Control.* Rohde & Schwarz GmbH & Co. KG, Muhldorfstrabe 15, 81671 Munchen, Germany, 1st edition, 2003.

[16] Robert W. Sebesta. *Concepts of Programming Languages.* Addison Wesley, 75 Ailington Street, Suite 300, Boston, MA, 5th edition, 2002.

[17] STROUSTRUP. *C++ PROGRAMMING LANGUAGE.* ADDISON WESLEY, 3rd edition, 2000.

[18] Andrew S. Tanenbaum. *Computer Networks.* Pearson Education, Inc., Upper Saddle River, New Jersey 07458, 4th edition, 2003.

[19] Agilent Technologies. *Agilent VISA - User's Guide*, 5th edition, 2001.

University of Southern Queensland
Faculty of Engineering & Surveying

Remote Access of Automated Test Equipment

A dissertation submitted by

Nathan John Hetherington

in fulfillment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering (Computer Systems) &
Bachelor of Information Technology (Applied Computer Science)

Submitted: October,2004

Appendix A

Project Specification

Appendix B

ASCII Character Set

ASCII Table

Decimal	ASCII	Binary
32	blank	00100000
33	!	00100001
34	"	00100010
35	#	00100011
36	$	00100100
37	%	00100101
38	&	00100110
40	(	00101000
41	)	00101001
42	*	00101010
44	,	00101100
45	-	00101101
46	.	00101110
65	A	01000001
66	B	01000010
67	C	01000011
68	D	01000100
69	E	01000101
70	F	01000110
71	G	01000111
72	H	01001000
73	I	01001001
74	J	01001010
75	K	01001011
76	L	01001100
77	M	01001101
78	N	01001110
79	O	01001111
80	P	01010000
81	Q	01010001
82	R	01010010
83	S	01010011
84	T	01010100
85	U	01010101
86	V	01010110
87	W	01010111
88	X	01011000
89	Y	01011001
90	Z	01011010

Decimal	ASCII	Binary
91	[	01011011
92	/	01011100
93	]	01011101
94	^	01011110
95	_	01011111
96	’	01100000
97	a	01100001
98	b	01100010
99	c	01100011
100	d	01100100
101	e	01100101
102	f	01100110
103	g	01100111
104	h	01101000
105	i	01101001
106	j	01101010
107	k	01101011
108	l	01101100
109	m	01101101
110	n	01101110
111	o	01101111
112	p	01110000
113	q	01110001
114	r	01110010
115	s	01110011
116	t	01110100
117	u	01110101
118	v	01110110
119	w	01110111
120	x	01111000
121	y	01111001
122	z	01111010
123	{	01111011
124	|	01111100
125	}	01111101
126	~	01111110

Appendix C

GPIB Interface Cards

GPIB

F

F-02 Lineup / Basics Knowledge

GPIB Bus master transmission / High performance GPIB F Series

F-04 Features Of GPIB F Series

F-05 Low Profile PCI Series

F-05 PCI Bus Series

F-06 Compact PCI Bus Series

F-06 PC Card Series

Standard Type

F-07 PCI Bus SERIES

F-07 PC Card SERIES

F-08 ISA Bus SERIES

[Lineup]

●PCI Bus / Low Profile PCI Bus

Name	IEE-488.2	Speed [bps]	Bus Master Transmison	FIFO Memory	Bus Analyer Function	Software		Page
						ACX-PAC(W32)	API-PAC(W32)	
High performance F Series for PCI / Low Profile PCI								
GP-IB(LPCI)F	○	1.5Mbyte/sec (Max.)	○	Sender: 2Kbyte Reveiver:2Kbyte	○	-	Attached	F-05
High performance F Series for PCI Bus								
GP-IB(PCI)F	○	1.5Mbyte/sec (Max.)	○	Sender: 2Kbyte Reveiver:2Kbyte	○	-	Attached	F-05
GP-IB(PCI)F	○	1.5Mbyte/sec (Max.)	○	Sender: 2Kbyte Reveiver:2Kbyte	○	-	Attached	F-05
Standard Series for PCI Bus								
GP-IB(PCI)	○	1.2Mbyte/sec (Max.)	-	Sender: 2Kbyte Reveiver:2Kbyte	○	○	Attached	F-05
GP-IB(PCI)L	○	120Kbyte/sec (Max.)	-	-	○	○	Attached	F-05

●Compact PCI Bus

Name	IEE-488.2	Speed [bps]	Bus Master Transmison	FIFO Memory	Bus Analyer Function	Software		Page
						ACX-PAC(W32)	API-PAC(W32)	
High performance F Series								
GP-IB(CPCI)F	○	1.5Mbyte/sec (Max.)	○	Sender: 2Kbyte Reveiver:2Kbyte	○	-	Attached	F-06

●PC Card

Name	IEE-488.2	Speed [bps]	Bus Master Transmison	FIFO Memory	Bus Analyer Function	Software		Page
						ACX-PAC(W32)	API-PAC(W32)	
High performance F Series for PCI / Low Profile PCI slot								
GP-IB(LPCI)F	○	1.5Mbyte/sec (Max.)	○	Sender: 2Kbyte Reveiver:2Kbyte	○	-	Attached	F-06
High performance F Series for PCI Bus								
GP-IB(PM)	○	50Kbyte/sec (Max.)	-	-	-	○	Attached	F-07

●ISA Bus

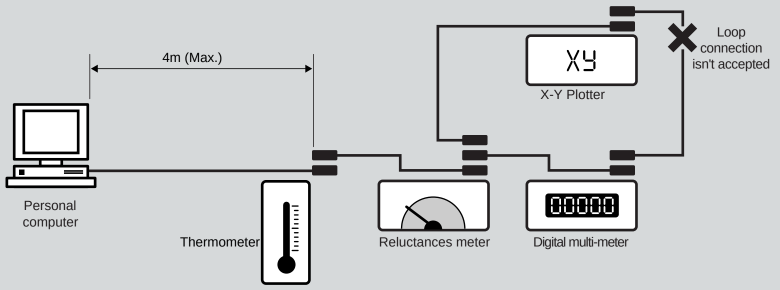
Name	IEE-488.2	IEE488.2	Speed [bps]	Bus Master Transmison	FIFO Memory	Bus Analyer Function	Software		Page
							ACX-PAC(W32)	API-PAC(W32)	
Standard Series									
GP-IB(PC)F	○	○	Using FIFO: 1Mbyte/sec (Reveiver) 700Kbyte/sec (Sender)	-	○	-	○	○	F-08
GP-IB(PC)L	○	○	Nomal: 120Kbyte/sec(Max.) DMA: 400Kbyte/sec(Max.)	-	-	-	○	○	F-08
GP-IB(PC)	○	-	DMA:300Kbyte/sec(Max.)	-	-	-	○	○	F-08

Basic Knowledge Concerning GPIB Communication

1. GPIB communication standards

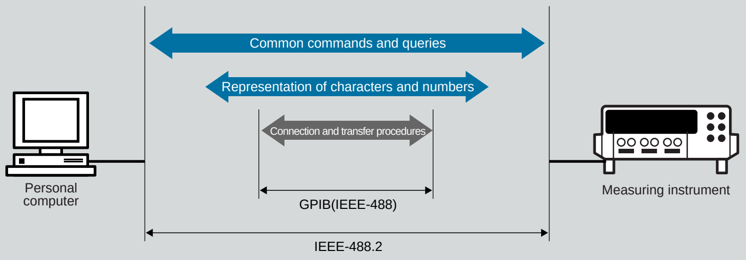
■GPIB (IEEE-488)

GPIB (General Purpose Interface Bus) was originally developed as an interface between a computer and a measurement device. HP-IB which was an in-house standard of Hewlett Packard was later approved by IEEE (Institute of Electrical and Electronic Engineers) to become the global standard and is presently adopted by many measurement devices. It is capable of connecting up to 15 devices. This standard is also called IEEE-488, IEEE-IB and IEC625, but they are practically the same as HP-IB.



■IEEE-488.2

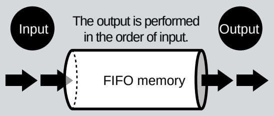
As a host protocol of IEEE-488.1, it provides additional rules concerning the grammar of character data and numeric representation as well as the commands and queries that can be commonly used among each device as a supplement to the transfer procedure stipulated in IEEE-488.1. Therefore, IEEE 488.2-compliant interface boards also satisfy the communication requirements provided in IEEE-488.1.


Common commands and queries
Representation of characters and numbers
Connection and transfer procedures
Personal computer
Measuring instrument
GPIB(IEEE-488)
IEEE-488.2

■Functions

Bus analyzer function
It analyses the data flowing on the line and monitors the status of each signal.

FIFO memory
FIFO is a storage device which stands for First In First Out. A board equipped with FIFO memory is capable of transmitting/receiving bulk data at high speeds.



Lineup/Basic Knowledge
High performance F Series
Feature
Low Profile PCI Bus
PCI Bus
Compact PCI Bus
PC Card
Standard Type
PCI Bus
PC Card
ISA Bus

Lineup/Basic Knowledge

High performance F Series

Feature

Low Profile PCI Bus

PCI Bus

Compact PCI Bus

PC Card

Standard Type

PCI Bus

PC Card

ISA Bus

[GPIB] Features of high-grade and high-speed type (GPIB F Series)

The new series of PC GPIB communication boards are equipped with all the desired features including IEEE-488.2 compliance, bus master high-speed data transmission and GPIB bus line analysis. The major features and functions of this series, include:

●Low Profile PCI bus supported : GP-IB(LPCI)F

●PCI bus supported : GP-IB(PCI)F, GP-IB(PCI)FL

●Compact PCI bus supported : GP-IB(CPCI)F

●PC Card (CardBus) supported : GP-IB(CB)F

1. Compliant with IEEE-488.2

Compliant with IEEE-488.2, it is capable of controlling various external devices which satisfy this standard.

2. Maximum transfer speed: 1.5Mbyte/sec

Maximum transfer speed is capable of communicating with a maximum transfer speed of 1.5Mbyte/sec.

3. Bus master transfer

Bus master transfer function allows for the transfer of bulk data between PC and the board without applying additional load on CPU.

4. 2Kbyte FIFO both for transmission and reception

2Kbyte FIFO is provided respectively for transmission and reception, allowing for high-speed transmission of small to large size data.

High-speed transmission is also possible with interface message using FIFO.

5. GPIB bus analyzer

Equipped with GPIB bus analyzer (excluding GP-IB(PCI)FL), it is capable not only of analyzing the signals which flow on GPIB bus but also of conducting signal analysis while GPIB communication is in progress on the board.

6. SPAS event (slave mode)

In addition to the conventional GPIB controller (μPD7210), event (SPAS) is also provided at the time of serial poll, allowing for highly flexible system configuration.

7. High-precision timer

The built-in high-precision application timer enables the precise time monitoring on Windows.

8. Stable supply over the long term

The unit is equipped with high-speed GPIB controller (μPD7210-upper compatible) developed by CONTEC. Therefore, stable supply over the long term is assured.

9. Diagnosis program

A diagnosis program is provided as a tool for supporting the system configuration. Diagnosis program is capable of conducting hardware operation check (interrupt / I/O access) and simple communication test with connected devices.

10. Driver library API-PAC(W32)

The standardly equipped driver library allows you to create Windows application software using various languages supporting Win32API function such as Visual Basic and Visual C/C++ in combination with LabVIEW.

11. Additional functions

●Line monitoring function

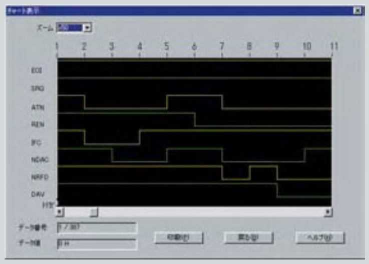
Line monitoring function is capable of reading the status of all the control lines (IFC, ATN, SRQ, REN, EOI, DAV, NRFD and NDAC) and latch data as well as the status of data line (DIO1 to DIO8) excluding GP-IB(PCI)FL.

●FIFO communication

FIFO memory provided on the board can be used for transmission and reception. Since the communication is controlled on the board side, high-speed communication is possible regardless of the CPU speed. The actual communication speed will be the speed of the slowest device in accordance with GPIB communication standard.

●Analyzer function (excluding GP-IB(PCI)FL)

By using the memory provided on the board, it can analyze the status change of all the lines on the GPIB cable. (Maximum 64K data can be fetched.) Analyzer function can be used to determine the point at which a failure has occurred or to check the data which flows on the line. This function can be used on the analyzer utility (Analyzer.exe).



Cable & Accessories

GPIB Cable

PCN-T02	2m
PCN-T04	4m

Double Shield UL

GPIB standard-compliant, this dedicated connection cable is noise-resistant and highly reliable.

GPIB Connector Adapter

CN-GP/C

This connector adapter is highly convenient when the expansion slot of a PC has an extended depth or when there is an interference with the cable from the neighboring board.

High-performance IEEE-488.2/GPIB board GP-IB(LPCI)F

NEW

FEATURES

- ●Compliant with IEEE-488.1 and IEEE-488.2
- ●Maximum transfer speed of 1.5Mbyte/sec
- ●Bus master transfer allows for the transfer of bulk data without applying additional load on CPU.
- ●2Kbyte FIFO is provided respectively for transmission and reception.
- ●Equipped with GPIB bus analyzer function.
- ●The unit is equipped with high-speed GPIB controller developed by CONTEC. Therefore, stable supply over the long term is assured.

PCI GPIB Bus Master

API function library attachment

■SPECIFICATIONS

Interface Type	IEEE488.1, IEEE488.2
Channels	1ch
Speed	1.5M byte/sec (Max.)
Data Format	8 parallel limes, 3 handshake lines
Signal logic	Negative logic Low level : 0.8V or less, High level : 2.0V or more
Interrupt	One interrupt request signal as INTA
I/O address	Any 128-byte boundary
Cable length betw een device	4m or less
Total cable length	20m or less
Connectable number of	Max. 15 devices
Power consumption (Max.)	+5VDC 400mA

Connecter	555139-1[AMP] or equivalent
PCI bus/ Dimension (mm)	32bit, 33MHz,5V/ 121.69(L)x63.41(H)
Option	
Software	-
Accessories	CN-GP/C
Cables/Connector	PCN-T02, PCN-T04

High-performance IEEE-488.2/GPIB board GP-IB(PCI)F

NEW

FEATURES

- ●Compliant with IEEE-488.1 and IEEE-488.2
- ●Maximum transfer speed of 1.5Mbyte/sec
- ●Bus master transfer allows for the transfer of bulk data without applying additional load on CPU.
- ●2Kbyte FIFO is provided respectively for transmission and reception.
- ●Equipped with GPIB bus analyzer function.
- ●The unit is equipped with high-speed GPIB controller developed by CONTEC. Therefore, stable supply over the long term is assured.

PCI GPIB Bus Master

API function library attachment

■SPECIFICATIONS

Interface Type	IEEE488.1, IEEE488.2
Channels	1ch
Speed	1.5Mbyte/sec (Max.)
Data Format	8 parallel limes, 3 handshake lines
Signal logic	Negative logic Low level : 0.8V or less, High level : 2.0V or more
Interrupt	One interrupt request signal as INTA
I/O address	Any 128-byte boundary
Cable length betw een device	4m or less
Total cable length	20m or less
Connectable number of	Max. 15 devices
Power consumption (Max.)	+5VDC 400mA

Connecter	555139-1[AMP] or equivalent
PCI bus/ Dimension (mm)	32bit, 33MHz, 5V/ 121.69(L)x63.41(H)
Option	
Software	-
Accessories	CN-GP/C
Cables/Connector	PCN-T02, PCN-T04

Low cost High-performance IEEE-488.2/GPIB board GP-IB(PCI)FL

NEW

FEATURES

- ●Compliant with IEEE-488.1 and IEEE-488.2
- ●Maximum transfer speed of 1.5Mbyte/sec
- ●Bus master transfer allows for the transfer of bulk data without applying additional load on CPU.
- ●2Kbyte FIFO is provided respectively for transmission and reception.
- ●The unit is equipped with high-speed GPIB controller developed by CONTEC. Therefore, stable supply over the long term is assured.

PCI GPIB Bus Master

API function library attachment

■SPECIFICATIONS

Interface Type	IEEE488.1, IEEE488.2
Channels	1ch
Speed	1.5Mbyte/sec (Max.)
Data Format	8 parallel limes, 3 handshake lines
Signal logic	Negative logic Low level : 0.8V or less, High level : 2.0V or more
Interrupt	One interrupt request signal as INTA
I/O address	Any 128-byte boundary
Cable length betw een device	4m or less
Total cable length	20m or less
Connectable number of	Max. 15 devices
Power consumption (Max.)	+5VDC 400mA

Connecter	555139-1[AMP] or equivalent
PCI bus/ Dimension (mm)	32bit, 33MHz, 5V/ 121.69(L)x63.41(H)
Option	
Software	-
Accessories	CN-GP/C
Cables/Connector	PCN-T02, PCN-T04

Lineup/Basic Knowledge
High performance F Series
Feature
Low Profile PCI Bus
PCI Bus
Compact PCI Bus
PC Card
Standard Type
PCI Bus
PC Card
ISA Bus

Lineup/Basic Knowledge
High performance F Series
Feature
Low Profile PCI Bus
PCI Bus
Compact PCI Bus
PC Card
Standard Type
PCI Bus
PC Card
ISA Bus

High-performance IEEE-488.2/GPIB board GP-IB(CPCI)F

Compact PCI GPIB Bus Master

API function library attachment

FEATURES
- ●Compliant with IEEE-488.1 and IEEE-488.2
- ●Maximum transfer speed of 1.5Mbyte/sec
- ●Bus master transfer allows for the transfer of bulk data without applying additional load on CPU.
- ●2Kbyte FIFO is provided respectively for transmission and reception.
- ●Equipped with GPIB bus analyzer function.
- ●The unit is equipped with high-speed GPIB controller developed by CONTEC. Therefore, stable supply over the long term is assured.

■SPECIFICATIONS

Interface Type	IEEE488.1, IEEE488.2
Channels	1ch
Speed	1.5Mbyte/sec (Max.)
Data Format	8 parallel limes, 3 handshake lines
Signal logic	Negative logic Low level : 0.8V or less, High level : 2.0V or more
Interrupt	One interrupt request signal as INTA
I/O address	Any 128-byte boundary
Cable length betw een device	4m or less
Total cable length	20m or less
Connectable number of	Max. 15 devices
Power consumption (Max.)	+5VDC 400mA

Connecter	555139-1[AMP] or equivalent
PCI bus/ Dimension (mm)	Compact PCI/3Ux4HP
Option	
Software	-
Accessories	CN-GP/C
Cables/Connector	PCN-T02, PCN-T04

CardBus High-performance IEEE-488.2/GPIB board GP-IB(CB)F

NEW

Card Bus GPIB Bus Master

API function library attachment

FEATURES
- ●Compliant with IEEE-488.1 and IEEE-488.2
- ●Maximum transfer speed of 1.5Mbyte/sec
- ●Bus master transfer allows for the transfer of bulk data without applying additional load on CPU.
- ●2Kbyte FIFO is provided respectively for transmission and reception.
- ●Equipped with GPIB bus analyzer function.
- ●The unit is equipped with high-speed GPIB controller developed by CONTEC. Therefore, stable supply over the long term is assured.

■SPECIFICATIONS

Interface Type	IEEE488.1, IEEE488.2
Channels	1ch
Speed	1.5Mbyte/sec (Max.)
Data Format	8 parallel limes, 3 handshake lines
Signal logic	Negative logic Low level : 0.8V or less, High level : 2.0V or more
Interrupt	One interrupt request signal as INTA
I/O address	Any 128-byte boundary
Cable length betw een device	4m or less
Total cable length	20m or less
Connectable number of	Max. 15 devices
Power consumption (Max.)	+5VDC 400mA

Length of an attached cable	2m
Card type	CardBus TYPE II
Weight	250g (include a cable)
Option	
Software	-
Accessories	CN-GP/C
Cables/Connector	PCN-T02, PCN-T04

IEEE-488.2/GPIB Interface Board

GP-IB(PCI)

PCI GPIB Memory on board CE

API function library attachment [API-PAC(W32)]

FEATURES

- ●Data can be transmitted at a maximum transmission rate of 1.2 Mbyte/sec
- ●1Mbyte of FIFO is built in for data transmission and reception, so the maximum capacity data communication can be conducted
- ●Contains a GPIB bus analyzer function

■SPECIFICATIONS

Interface Type	IEEE488.1, IEEE488.2
Channels	1ch
Speed	1.2Mbyte/sec (Max.)
Data Format	8 parallel limes, 3 handshake lines
Signal logic	Negative logic Low level : 0.8V or less, High level : 2.0V or more
Interrupt	One interrupt request signal as INTA
I/O address	Any 16-byte boundary
Cable length between device	4m or less
Total cable length	20m or less
Connectable number of device	Max. 15 devices
Power consumption (Max.)	+5VDC 970mA

Connector		555139-1[AMP] or equivalent
PCI bus/		32bit, 33MHz, 5V/
Dimension (mm)		121.69(L)x106.68(H)
Option		
	Software	ACX-PAC(W32)BP, ACX-PAC(W32)AP, SUPPORT-PAC(PC)202 Ver.2.30 upper,
	Accessories	CN-GP
	Cables/Connector	PCN-02, PCN-04

IEEE-488.2/GPIB Interface Board

GP-IB(PCI)L

PCI GPIB CE

API function library attachment [API-PAC(W32)]

FEATURES

- ●Conforming to the IEEE-488.2 standard, the GP-IB(PCI)L can exchange signals with a variety of external devices compliant with the standard
- ●The board can use the CONTEC FPGA (uPD7210C compatible) as a GPIB controller for long-term stable supply

■SPECIFICATIONS

Interface Type	IEEE488.1, IEEE488.2
Channels	1ch
Speed	120Kbyte/sec (Max.)
Data Format	8 parallel limes, 3 handshake lines
Signal logic	Negative logic L-Level: 0.8V or less H-Level: 2.0V or more
Controller chip	CONTEC original FPGA (uPD7210C compatible)
Interrupt	One interrupt request signal as INTA
I/O address	Any 32-byte boundary
Cable length between device	4m or less
Total cable length	20m or less
Connectable number of device	Max. 15 devices

Power consumption (Max.)		+5VDC 300mA
Connector		555139-1[AMP] or equivalent
PCI Bus/		32bit, 33MHz, 5V/
Dimension (mm)		121.69(L)x106.68(H)
Option		
	Software	ACX-PAC(W32)BP Ver.2.1 upper, ACX-PAC(W32)AP Ver.2.1 upper, SUPPORT-PAC(PC)202 Ver.2.40 upper,
	Accessories	CN-GP
	Cables/Connector	PCN-02, PCN-04

IEEE-488.2/GPIB Interface Card

GP-IB(PM)

PC Card GPIB CE

API function library attachment [API-PAC(W32)]

FEATURES

- ●Conforms to IEEE-488.2 specifications
- ●Data transfer can be performed at a maximum rate of 50Kbytes/sec

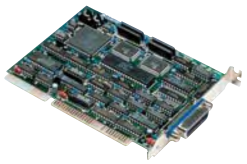
■SPECIFICATIONS

Channels	1
Type	IEEE488.1, IEEE488.2
Interface type	8 parallel limes, 3 handshake lines
Speed	50Kbyte/sec (Max.)
Signal logic	Low logic level : 0.8V or less, High logic level : 2.0V or more
Total cable length	20m
Cable length between device	4m
Connectable number of device(Max.)	Max. 15 devices
Interrupt Request Level	One of IRQ 3-7, 9-12, 14 or 15

I/O address		Any 16-byte boundary
Power consumption (Max.)		DC+5V 100mA
Operating Temp./Humidity		0-50℃, 20-90%RH (no condensation)
Length of an attached cable		2m
Card type		PCMCIA 2.1/JEIDA 4.2, 16-bit PC Card, JEIDA Type II
Weight		250g (include a cable)
Option		
	Software	ACX-PAC(W32)BP, ACX-PAC(W32)AP, SUPPORT-PAC(PC)202
	Cables/Connector	PCN-02, PCN-04

[GPIB] ISA Bus

For the optional items, please refer to Page G-01 (support software).

ISA Model / Specification		IEEE-488/GPIB Interface Board GP-IB(PC)L	IEEE-488 Interface Board GP-IB(PC)F	DMA IEEE-488/GPIB Interface Board GP-IB(PC)
				
Interface type		IEEE488.1, IEEE488.2		IEEE-488.1
Channels		1ch		
Speed		[DMA mode] 400Kbyte/sec (Max.)	[DMA]Sender/Receiver: 60Kbyte/sec, [FIFO]Sender:700Kbyte/sec, Reciver:1Mbyte/sec	[DMA]Sender/Receiver: 300Kbyte/sec
Data type		8 parallel lines, 3 handshake lines		
Signal logic		Low level: 0.8V or less, High level: 2.0V or more (Negative logic)	TTL-level (Negative logic)	
DMA channels		-	CH1~CH3 (jumper selectable)	
Controller chip		CONTEC original FPGA (μPD7210C compatible)	PD7210 or equivalent	
Interrupt		1 interrupt request signal as INTA	One of IRQ 3~7, 9~12, 14 or 15	One of IRQ 3~7, 9
I/O address		Any of 32-byte boundary	Any of 17-byte boundary	Any of 9-byte boundary
Cable length between devices		4m or less		
Total cable length		20m or more		
Connectable devices		15 devices (Max.)		
Power consumption (Max.)		+5VDC 350mA	+5VDC 750mA	+5VDC 400mA
Connector		555139-2(AMP)	57LE-20240-77C0D35G [DDK] or equivalent	
Bus / Dimension (mm)		AT Bus / 163.0(L) x 107.0(H)	AT Bus / 163.0(L) x 122.0(H)	XT Bus / 116.0(L) x 107.0(H)
Option	**Software**	ACX-PAC(W32)BP, ACX-PAC(W32)AP, API-PAC(W32), SUPPORT-PAC(PC)202, SUPPORT-PAC(98)202	ACX-PAC(W32)BP, ACX-PAC(W32)AP, API-PAC(W32), SUPPORT-PAC(PC)202	
	Accessories	CN-GP/C		
	Cables / Connector	PCN-T02, PCN-T04		
CE marking		-	○	○

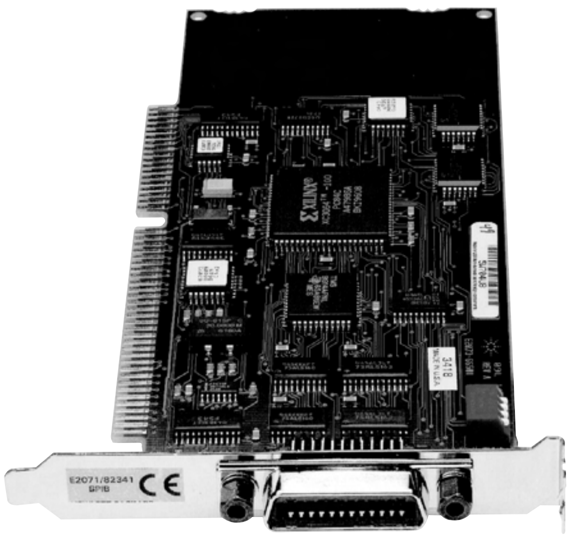
Notes: Software of option is required.

Agilent 82341C High-Performance ISA GPIB Interface for Windows®

Data Sheet

- Built-in buffering for speed
- Agilent VEE compatibility
- BASIC for Windows compatibility
- Microsoft languages compatibility
- SICL/VISA support

Description



Agilent 82341C

The Agilent Technologies 82341C is a **low cost, high-performance IEEE-488 interface and software for ISA-based PCs**. With the 82341C it is easy to access and control instruments and exchange data. This is a **high-speed, 16-bit card** with built-in buffering, which de-couples GPIB transfers from ISA bus transfers. Buffering provides I/O and system performance that is superior to direct memory access (DMA)–up to 750 KB/s.

The 82341C includes SICL and VISA I/O software for Windows 95/98/NT/2000.

The GPIB interface card plugs into an ISA slot in the backplane of your PC. Via a GPIB cable, this card connects to GPIB instruments.

For VXI applications a GPIB cable connects the card to the command module in Slot 0 of the VXI mainframe.

Refer to the Agilent Technologies Website for instrument driver availability and downloading instructions, as well as for recent product updates, if applicable.

Agilent Technologies

Product Specifications

Operating system:	Windows 95/98/NT/2000
Controller:	PC
I/O library:	SICL / VISA
Backplane:	ISA
Max. I/O speed:	750 KB/s
Buffering:	Built-in
Languages:	Agilent VEE, C/C++, BASIC for Windows, Visual Basic

Ordering Information

Description	Product No.
High-Performance ISA GPIB Interface for Windows	82341C
Add Manual Set	82341C 0B1
Upgrade from 82335B	82341C AGE

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

Agilent Technologies' Test and Measurement Support, Services, and Assistance

Agilent Technologies aims to maximize the value you receive, while minimizing your risk and problems. We strive to ensure that you get the test and measurement capabilities you paid for and obtain the support you need. Our extensive support resources and services can help you choose the right Agilent products for your applications and apply them successfully. Every instrument and system we sell has a global warranty. Support is available for at least five years beyond the production life of the product. Two concepts underlie Agilent's overall support policy: "Our Promise" and "Your Advantage."

Our Promise
Our Promise means your Agilent test and measurement equipment will meet its advertised performance and functionality. When you are choosing new equipment, we will help you with product information, including realistic performance specifications and practical recommendations from experienced test engineers. When you use Agilent equipment, we can verify that it works properly, help with product operation, and provide basic measurement assistance for the use of specified capabilities, at no extra cost upon request. Many self-help tools are available.

Your Advantage
Your Advantage means that Agilent offers a wide range of additional expert test and measurement services, which you can purchase according to your unique technical and business needs. Solve problems efficiently and gain a competitive edge by contracting with us for calibration, extra-cost upgrades, out-of-warranty repairs, and on-site education and training, as well as design, system integration, project management, and other professional engineering services. Experienced Agilent engineers and technicians worldwide can help you maximize your productivity, optimize the return on investment of your Agilent instruments and systems, and obtain dependable measurement accuracy for the life of those products.

By internet, phone, or fax, get assistance with all your test & measurement needs.

Online assistance:
www.agilent.com/find/assist

Phone or Fax
United States:
(tel) 1 800 452 4844

Canada:
(tel) 1 877 894 4414
(fax) (905) 282 6495

China:
(tel) 800 810 0189
(fax) 1 0800 650 0121

Europe:
(tel) (31 20) 547 2323
(fax) (31 20) 547 2390

Japan:
(tel) (81) 426 56 7832
(fax) (81) 426 56 7840

Korea:
(tel) (82 2) 2004 5004
(fax) (82 2) 2004 5115

Latin America:
(tel) (305) 269 7500
(fax) (305) 269 7599

Taiwan:
(tel) 080 004 7866
(fax) (886 2) 2545 6723

Other Asia Pacific Countries:
(tel) (65) 375 8100
(fax) (65) 836 0252
Email: tm_asia@agilent.com

Agilent Technologies

Appendix D

HTML Pages

D.1 Function Generator

```
<html>

<head><title align=center> 33120A 15Mhz Function/Arbitrary waveform genterator </title>
  <h1 align=center> 33120A Function/Arbitrary Waveform Generator </h1>
</head>
<BODY>
<FORM action=/cgi−bin/cgiprogram1.exe method=post>
   <TABLE align="center" border="1" width="668" cellpadding="3" id="table1">

<center> Select Waveform Type :
<br>
<input type="radio" name="group1" value="Sine" checked> Sine<br>
<input type="radio" name="group1" value="Triangular"> Triangular<br>
<input type="radio" name="group1" value="Square"> Square<br>
<input type="radio" name="group1" value="Ramp"> Ramp<br>
<hr>


      <TR>
           <TD width="222">Frequency :
         <input type="text" name="freq1" size="20">
         </TD>
           <TD width="300">Peak To Peek Voltage :
         <input type="text" name="ptpvolt" size="20">
         </TD>
         <TD width="222">Offset :
         <input type="text" name="Offs" size="20">
         </TD>
      </TR>
   </TABLE>

</CENTER>
<CENTER>
<INPUT type=submit value="Press to Submit">
</CENTER>
</FORM>
<FORM action=/cgi-bin/cgipost.exe method=post>
   <TABLE align="center" border="1" width="668" cellpadding="3" id="table1">

<CENTER> Sweep over a range of frequencies :
<BR>

      <TR>
          <TD width="222">From :
        <input type="text" name="From" size="20">
        </TD>
          <TD width="300">To :
        <input type="text" name="To" size="20">
        </TD>
        <TD width="222">Step :
        <input type="text" name="Step" size="20">
        </TD>
      </TR>
   </TABLE>

</CENTER>
<CENTER>
<INPUT type=submit value="Press to Submit">
</CENTER>
</FORM>
</BODY>

</html>
```

D.2 Switch Unit

```
<html>

<head><title align=center> 34970A Data Acquisition/Switch Unit </title>
  <h1 align=center> 34970A Data Acquisition/Switch Unit </h1>
</head>
<BODY>

<FORM action=/cgi-bin/34970open.exe method=post>
<TABLE align="center" border="1" width="668" cellpadding="3" id="table1">
```

```
   <CENTER>
   <TR>
         <TD width="222">Open switches(For multiple seperate with ','):
       <input type="text" name="freq1" size="20">
       </TD>

     </TR>
   </CENTER>
   </TABLE>
<CENTER>
<INPUT type=submit value="Press to Submit">
</CENTER>
</FORM>

<FORM action=/cgi-bin/34970close.exe method=post>
<TABLE align="center" border="1" width="668" cellpadding="3" id="table1">
   <CENTER>
   <TR>
         <TD width="222">Close switches(For multiple seperate with ','):
       <input type="text" name="freq1" size="20">
       </TD>

     </TR>
   </CENTER>
   </TABLE>
<CENTER>
<INPUT type=submit value="Press to Submit">
</CENTER>
</FORM>

</BODY>

</html>
```

D.3 Power Supply

```
<html>

<head><title align=center> 6624A System DC Power Supply </title>
  <h1 align=center> 6624A System DC Power Supply </h1>
</head>

<BODY>
<FORM action=/cgi-bin/6624a.exe method=post>

<center> Select Supply Number :
```

```
<br>
<input type="radio" name="group1" value="1" checked> 1<br>
<input type="radio" name="group1" value="2">2<br>
<input type="radio" name="group1" value="3"> 3<br>
<input type="radio" name="group1" value="4"> 4<br>
<hr>


   <TABLE align="center" border="1" width="668" cellpadding="3" id="table1">
   <CENTER>
   <TR>
         <TD width="222">Voltage:
      <input type="text" name="freq1" size="20">
      </TD>

     </TR>
   </CENTER>
   </TABLE>

<CENTER>
<INPUT type=submit value="Press to Submit">
</CENTER>
</FORM>
</BODY>

</html>
```

D.4 Oscilloscope

```
<html>

<head><title align=center> 54602B Osilliscope </title>
  <h1 align=center> 54602B Osilliscope </h1>
</head>
<body>
<center>
<FORM action=/cgi−bin/54602b.exe method=post>
   <TABLE align="center" border="1" width="668" cellpadding="3" id="table1">

<hr>
<center> Select Desired Channel :
<br>
<input type="radio" name="channel" value="CHANNEL1" checked> Channel 1
<input type="radio" name="channel" value="CHANNEL2"> Channel 2
<input type="radio" name="channel" value="CHANNEL3"> Channel 3
<input type="radio" name="channel" value="CHANNEL4"> Channel 4
```

```
</center>

<hr>
<center> Select Horizontal Scale (Time/div) :
<br>
<input type="radio" name="time" value="5s" checked> 5s
<input type="radio" name="time" value="2s"> 2s
<input type="radio" name="time" value="1s"> 1s<br>
<input type="radio" name="time" value=500ms"> 500ms
<input type="radio" name="time" value="200ms"> 200ms
<input type="radio" name="time" value="50ms"> 50ms
<input type="radio" name="time" value="20ms"> 20ms
<input type="radio" name="time" value="10ms"> 10ms
<input type="radio" name="time" value="5ms"> 5ms
<input type="radio" name="time" value="2ms"> 2ms
<input type="radio" name="time" value="1ms"> 1ms<br>
<input type="radio" name="time" value="500us"> 500us
<input type="radio" name="time" value="200us"> 200us
<input type="radio" name="time" value="100us"> 100us
<input type="radio" name="time" value="50us"> 50us
<input type="radio" name="time" value="20us"> 20us
<input type="radio" name="time" value="10us"> 10us
<input type="radio" name="time" value="5us"> 5us
<input type="radio" name="time" value="2us"> 2us
<input type="radio" name="time" value="1us"> 1us<br>
<input type="radio" name="time" value="500ns"> 500ns
<input type="radio" name="time" value="200ns"> 200ns
<input type="radio" name="time" value="100ns"> 100ns
<input type="radio" name="time" value="50ns"> 50ns
<input type="radio" name="time" value="20ns"> 20ns
<input type="radio" name="time" value="10ns"> 10ns
<input type="radio" name="time" value="5ns"> 5ns
<input type="radio" name="time" value="2ns"> 2ns
<input type="radio" name="time" value="1ns"> 1ns<br>
<hr>
<center> Select Vertical Scale (Volts/div) :
<br>
<input type="radio" name="Vscale" value="5V" checked> 5V
<input type="radio" name="Vscale" value="2V"> 2V
<input type="radio" name="Vscale" value="1mV"> 1V<br>
<input type="radio" name="Vscale" value="500mV"> 500mV
<input type="radio" name="Vscale" value="200mV"> 200mV
<input type="radio" name="Vscale" value="100mV"> 100mV
<input type="radio" name="Vscale" value="50mV"> 50mV
<input type="radio" name="Vscale" value="20mV"> 20mV
<input type="radio" name="Vscale" value="10mV"> 10mV<br>
<input type="radio" name="Vscale" value="5mV"> 5mV
<input type="radio" name="Vscale" value="2mV"> 2mV
```

```
<input type="radio" name="Vscale" value="1mV"> 1mV<br>
<hr>
</center>

<INPUT type=submit value="Press to Submit">
</CENTER>
</FORM>

</body>


</html>
```

Appendix E

CGI Applications

E.1 Basic CGI Applications

E.1.1 CGI get Method

```
/* cgiget.c
 * CGI using GET method (environment variables)
 *
 * See cgiget.html for use in conjunction with a HTML file.
 *
 * Sample output:
 *
    QUERY˙STRING -¿ TEXT˙LINE˙ONE=some+text&amp;TEXT˙LINE˙TWO=default+value&amp;
                  TEXT˙LINE˙THREE=1234&amp;PASSWORD˙FIELD=secret
    SERVER˙SOFTWARE -¿ Apache/1.3.19 (Win32)
    SERVER˙NAME -¿ estaff100.eng.usq.edu.au
    SERVER˙PROTOCOL -¿ HTTP/1.1
    GATEWAY˙INTERFACE -¿ CGI/1.1
    PATH˙INFO is undefined
    REMOTE˙HOST is undefined
    SERVER˙USER is undefined
    REQUEST˙METHOD -¿ GET
    REMOTE˙ADDR -¿ 139.86.65.100
    HTTP˙ACCEPT -¿ image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
                  application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint,
    HTTP˙USER˙AGENT -¿ Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)
    HTTP˙REFERRER is undefined
 *
 * This code only shows some of the CGI environment variables -
 * see cgishow.c for a complete list.
 *
 * John Leis Aug 1998
 * Revised  June 2001
 */

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

#define  MAX_CGILEN  2048
#define  TMP_BUFLEN  512

// holds the contents of the http output (not including the header)
static char contentBuf[MAX_CGILEN+1];

int     main()
{
    char   *pName, *pValue;
    int    contentLen;
    char   tmpBuf[TMP_BUFLEN+1];


    // buffer which contains the content, ie HTML output
    contentBuf[0] = '\0';

    strcat(contentBuf, "<HTML>\n");
    strcat(contentBuf, "<BODY>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "  <I> cgiget.c V 1.1 John Leis </I> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "  <H2> Here's the result: </H2> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");

    // retrieve environment variable
    pName = "QUERY_STRING";
    pValue = getenv(pName);
    if( pValue )
    {
        // environment variable is defined
        sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
        strcat(contentBuf, tmpBuf);
    }
    else
    {
```

```
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "SERVER_SOFTWARE";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "SERVER_NAME";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "SERVER_PROTOCOL";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
```

```
strcat(contentBuf, "  <BR>\n");

pName = "GATEWAY_INTERFACE";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "PATH_INFO";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "REMOTE_HOST";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "SERVER_USER";
```

```
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "REQUEST_METHOD";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "REMOTE_ADDR";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "HTTP_ACCEPT";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "HTTP_USER_AGENT";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");

pName = "HTTP_REFERRER";
pValue = getenv(pName);
if( pValue )
{
    // environment variable is defined
    sprintf(tmpBuf, "  <B> %s -> %s </B> \n", pName, pValue);
    strcat(contentBuf, tmpBuf);
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}
strcat(contentBuf, "  <BR>\n");


// trailer
strcat(contentBuf, "</P>\n\n");
strcat(contentBuf, "</BODY>\n");
strcat(contentBuf, "</HTML>\n");

// now create the output according to the HTTP protocol
```

```

    // initial string specifies content text/plain, text/html etc
    printf("Content-type: text/html\n");

    // content length
    contentLen = strlen(contentBuf);
    printf("Content-length: %d\n", contentLen);

    // **** blank line is important here (separates header &amp; body) ****
    printf("\n");

    // now the content itself
    printf("%s", contentBuf);

    return 0;
}

```

E.1.2 CGI post Method

```
/* cgipost.c
 * CGI using POST method (standard input/output)
 * Note that this method is different to the GET method –
 * the input comes from standard input (stdin), and
 * not the environment variable QUERY_STRING
 *
 * See cgipost.html for use in conjunction with a HTML file.
 *
 * Example output:
    cgipost.c V 1.1 John Leis

    Here's the result:

    CONTENT_LENGTH = 98
    data read:
    TEXT_LINE_ONE=test&amp;TEXT_LINE_TWO=default+value&amp;TEXT_LINE_THREE=0123&amp;PASSWORI
 *
 *
 * John Leis
 * Revised June 2001
 */

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

#include <string.h>
```

```
#include <unistd.h>


#define   MAX_CGILEN  2048
#define   TMP_BUFLEN  512
#define   READ_BUFLEN 512

// holds the contents of the http output (not including the header)
static char contentBuf[MAX_CGILEN+1];

#ifndef   STDIN
#define   STDIN   0
#endif    STDIN

#ifndef   STDOUT
#define   STDOUT 1
#endif    STDOUT

int      main()
{
    int      bytesRead, bytesLeft;
    char     *pName, *pValue;
    int      contentLen;
    char     tmpBuf[TMP_BUFLEN+1];
    char     readBuf[READ_BUFLEN+1];


    // buffer which contains the content, ie HTML output
    contentBuf[0] = '\0';

    strcat(contentBuf, "<HTML>\n");
    strcat(contentBuf, "<BODY>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "   <I> cgipost.c V 1.1 John Leis </I> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "   <H2> Here's the result: </H2> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");

    // for testing only
    //sprintf(tmpBuf, "20");
    //setenv("CONTENT_LENGTH", tmpBuf, 1);

    pName = "CONTENT_LENGTH";
```

```
pValue = getenv(pName);
if( pValue )
{
    sprintf(tmpBuf, "  <B> %s = %s </B><BR>\n", pName, pValue);
    strcat(contentBuf, tmpBuf);

    bytesLeft = atoi(pValue);
    do
    {
        bytesRead = read(STDIN, &amp;readBuf[0], READ_BUFLEN);

        // null-terminate
        if( bytesRead > 0 )
        {
            readBuf[bytesRead] = '\0';
            bytesLeft -= bytesRead;
        }
    } while( (bytesLeft > 0) &amp;&amp; (bytesRead > 0) );

    if( bytesRead > 0 )
    {
        sprintf(tmpBuf, "  <B> data read: </B>\n");
        strcat(contentBuf, tmpBuf);

        strcat(contentBuf, "   <BR>\n");
        strcat(contentBuf, "   <I>\n");

        sprintf(tmpBuf, "  %s\n", readBuf);
        strcat(contentBuf, tmpBuf);
        strcat(contentBuf, "   </I>\n");
    }
    else
    {
        sprintf(tmpBuf, "  <B> no data read </B> \n");
        strcat(contentBuf, tmpBuf);
    }
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}

// trailer
strcat(contentBuf, "</P>\n\n");
strcat(contentBuf, "</BODY>\n");
strcat(contentBuf, "</HTML>\n");

```

```

    // now create the output according to the HTTP protocol

    // initial string specifies content text/plain, text/html etc
    printf("Content-type: text/html\n");

    // content length
    contentLen = strlen(contentBuf);
    printf("Content-length: %d\n", contentLen);

    // **** blank line is important here (separates header &amp; body) ****
    printf("\n");

    // now the content itself
    printf("%s", contentBuf);

    return 0;
}
```

E.2 CGI Programs for Devices

E.2.1 Function Generator

```
// This is a CGI program to change settings on the 33120a Function Generator
/* 33120A.c
 * CGI using POST method (standard input/output)
 * Note that this method is different to the GET method –
 * the input comes from standard input (stdin), and
 * not the environment variable QUERY_STRING
 *
 * See cgipost.html for use in conjunction with a HTML file.
 *
 * Example output:


    Here's the result:

    CONTENT_LENGTH = 98
    data read:
    TEXT_LINE_ONE=test&TEXT_LINE_TWO=default+value&TEXT_LINE_THREE=0123&PASSWORD_FIELD=secr
 *
 *
 *
 */

```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sicl.h"

#define DEVICE_ADDRESS "hpib7,10"


#define   MAX_CGILEN  2048
#define   TMP_BUFLEN  512
#define   READ_BUFLEN 512

// holds the contents of the http output (not including the header)
static char contentBuf[MAX_CGILEN+1];

#ifndef   STDIN
#define   STDIN   0
#endif

#ifndef   STDOUT
#define   STDOUT 1
#endif

// Function Prototypes
void getVariables(char *readBuf);

// Variables for changes to equiptment
char Frequency[7];
char PtPVolts[5];
char WavType[12];
char Offset[5];
char desired[30];
char actual[30];

int      main()
{
        FILE   *Out;
    int      bytesRead, bytesLeft;
    char     *pName, *pValue;
    int      contentLen;
    char     tmpBuf[TMP_BUFLEN+1];
    char     readBuf[READ_BUFLEN+1];
        INST id;
        char buff[256];

    // Install a default SICL error handler that logs an error message and
        // exits.  On Windows 95 view messages with the SICL Message Viewer,
        // and on Windows NT
        ionerror(I_ERROR_EXIT);

        // Open a device session using the DEVICE_ADDRESS
        id=iopen(DEVICE_ADDRESS);
        // Write the *RST string (and send an EOI indicator) to put the instrument
    // in a known state.
    iprintf(id, "*RST\n");
        ipromptf(id, "*IDN?\n", "%t", buff);
        //printf("Device = %s\n",buff);

        // Open the output file
        Out=fopen("output.txt","w");

    // buffer which contains the content, ie HTML output
    contentBuf[0] = '\0';

    strcat(contentBuf, "<HTML>\n");
    strcat(contentBuf, "<BODY>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "  <I> 33120.c V 1.1 Nathan Hetherington </I> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "  <H2> Here's the result: </H2> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");

    // for testing only
    //sprintf(tmpBuf, "20");
    //setenv("CONTENT_LENGTH", tmpBuf, 1);

    pName = "CONTENT_LENGTH";
    pValue = getenv(pName);



    if( pValue )
    {
        sprintf(tmpBuf, "  <B> %s = %s </B><BR>\n", pName, pValue);
        strcat(contentBuf, tmpBuf);

        bytesLeft = atoi(pValue);
        do
        {
            bytesRead = read(STDIN, &readBuf[0], READ_BUFLEN);

            // null-terminate
            if( bytesRead > 0 )
            {
                readBuf[bytesRead] = '\0';
                bytesLeft -= bytesRead;
            }
        } while( (bytesLeft > 0) && (bytesRead > 0) );

        if( bytesRead > 0 )
        {
            sprintf(tmpBuf, "  <B> data read: </B>\n");
            strcat(contentBuf, tmpBuf);

            strcat(contentBuf, "   <BR>\n");
            strcat(contentBuf, "   <I>\n");

            sprintf(tmpBuf, "  %s\n", readBuf);
            strcat(contentBuf, tmpBuf);
            strcat(contentBuf, "   </I>\n");
        }
        else
        {
            sprintf(tmpBuf, "  <B> no data read </B> \n");
            strcat(contentBuf, tmpBuf);
        }
    }
    else
    {
        sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
        strcat(contentBuf, tmpBuf);
    }




    // Manipulate the input string
        fprintf(Out,"%s\n",readBuf);
        getVariables(readBuf);
        fprintf(Out,"WavType value = %s\n",WavType);
        fprintf(Out,"Frequency value = %s\n",Frequency);
        fprintf(Out,"Point to Point voltage value = %s\n",PtPVolts);
    fprintf(Out,"Offset voltage value = %s\n",Offset);


        // Set the Function generator
        if((strcmp(WavType,"Sine")==0)){
                fprintf(Out,"Setting a Sine waveform!\n");
            iprintf(id,"APPL:SIN %s,%s, %s\n",Frequency,PtPVolts,Offset);

            // Created the desired string
            strcat(desired, "APPL:SIN ");
            strcat(desired,Frequency);
            strcat(desired," , ");
            strcat(desired,PtPVolts);
            strcat(desired," , ");
            strcat(desired,Offset);
            strcat(desired,"\n");

    }else if((strcmp(WavType,"Triangular")==0)){
            fprintf(Out,"Setting a Tringular waveform!\n");
            iprintf(id,"APPL:TRI %s,%s, %s\n",Frequency,PtPVolts,Offset);

            // Created the desired string
            strcat(desired, "APPL:TRI ");
            strcat(desired,Frequency);
            strcat(desired," , ");
            strcat(desired,PtPVolts);
            strcat(desired," , ");
            strcat(desired,Offset);
            strcat(desired,"\n");

    }else if((strcmp(WavType,"Square")==0)){
            fprintf(Out,"Setting a Square waveform!\n");
            iprintf(id,"APPL:SQU %s,%s, %s\n",Frequency,PtPVolts,Offset);

            // Created the desired string
            strcat(desired, "APPL:SQU ");
            strcat(desired,Frequency);
            strcat(desired," , ");
            strcat(desired,PtPVolts);
            strcat(desired," , ");
            strcat(desired,Offset);
            strcat(desired,"\n");

    }else{
            fprintf(Out,"Setting a Ramp waveform!\n");
            iprintf(id,"APPL:RAMP %s,%s, %s\n",Frequency,PtPVolts,Offset);

            // Created the desired string
            strcat(desired, "APPL:RAMP ");
            strcat(desired,Frequency);
            strcat(desired," , ");
            strcat(desired,PtPVolts);
            strcat(desired," , ");
            strcat(desired,Offset);
```

```
            strcat(desired,"\n");

    }



iprom t(id,"APPL?\n","%t",actual)

if(strcmp(actual,desired)==0){
    strcat(contentBuf, "<P> <H1> Changes Successful !!!!!!</H1></P>\n");
}else{
    strcat(contentBuf, "<P> <H1> Changes Unsuccessful !!!!!!</H1></P>\n");
}

// trailer
strcat(contentBuf, "</P>\n\n");
strcat(contentBuf, "</BODY>\n");
strcat(contentBuf, "</HTML>\n");


// now create the output according to the HTTP protocol

// initial string specifies content text/plain, text/html etc
printf("Content-type: text/html\n");

// content length
contentLen = strlen(contentBuf);
printf("Content-length: %d\n", contentLen);

// **** blank line is important here (separates header & body) ****
printf("\n");

// now the content itself
printf("%s", contentBuf);




    // Close files
    fclose(Out);

    //iclose(id);

    // For WIN16 programs, call _siclcleanup before exiting to release
// resources allocated by SICL for this application. This call is a
// no-op for WIN32 programs.
//_siclcleanup();

return 0;
```

```
}
/////////////////////////////////////////////////////////////////////////////////
// Function Name: getVariables(char *Buffer)
//
//
/////////////////////////////////////////////////////////////////////////////////
void getVariables(char *Buffer){

        int index=0,count=0;


        while(Buffer[index]!= '='){
                index++;
    }
        index++;
        while(Buffer[index] != '&'){
                WavType[count]=Buffer[index];
                index++;
                count++;
        }
        count++;
        index++;
        WavType[count]='\0';
    printf("WavType value = %s\n",WavType);

        while(Buffer[index]!= '='){
                index++;
        }
        count = 0;
        index++;
        while(Buffer[index] != '&'){
                Frequency[count]=Buffer[index];
                index++;
                count++;
        }
        count++;
        Frequency[count]='\0';
    printf("Frequency value = %s\n",Frequency);
        while(Buffer[index]!= '='){
                index++;
        }
        count = 0;
        index++;
        while(Buffer[index] != '&'){
                PtPVolts[count]=Buffer[index];
                index++;
                count++;
        }
        count++;
        PtPVolts[count]='\0';
    printf("Point to Point voltage value = %s\n",PtPVolts);

        while(Buffer[index]!= '='){
                index++;
        }


        count = 0;
        index++;
        while(Buffer[index] != '\0'){
                Offset[count]=Buffer[index];
                index++;
                count++;
        }
        count++;
        Offset[count]='\0';
        printf("Offset voltage value = %s \n",Offset);
}
```

E.2.2 Oscilloscope

```
// This is the first draft of the cgi program for the 54602B Osilloscope
/* 54602.c
 * CGI using POST method (standard input/output)
 * Note that this method is different to the GET method -
 * the input comes from standard input (stdin), and
 * not the environment variable QUERY_STRING
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sicl.h"

#define DEVICE_ADDRESS "hpib7,7"


#define   MAX_CGILEN  2048
#define   TMP_BUFLEN  512
#define   READ_BUFLEN 512

// holds the contents of the http output (not including the header)
static char contentBuf[MAX_CGILEN+1];
```

```
#ifndef   STDIN
#define   STDIN   0
#endif

#ifndef   STDOUT
#define   STDOUT 1
#endif

// Function Prototypes
void getVariables(char *readBuf);

// Variables for changes to equiptment
char Time[7];
char Vscale[5];
char Channel[10];
char actual[30];
char desired[30]

int       main()
{
        FILE   *Out;
    int       bytesRead, bytesLeft;
    char     *pName, *pValue;
    int       contentLen;
    char     tmpBuf[TMP_BUFLEN+1];
    char     readBuf[READ_BUFLEN+1];
        INST id;
        char buff[256];

    // Install a default SICL error handler that logs an error message and
        // exits.  On Windows 95 view messages with the SICL Message Viewer,
        // and on Windows NT
        ionerror(I_ERROR_EXIT);

        // Open a device session using the DEVICE_ADDRESS
        id=iopen(DEVICE_ADDRESS);
        // Write the *RST string (and send an EOI indicator) to put the instrument
        // in a known state.
        iprintf(id, "*RST\n");
        ipromptf(id, "*IDN?\n", "%t", buff);
        printf("Device = %s\n",buff);

        // Open the output file
        Out=fopen("54602b.txt","w");

    // buffer which contains the content, ie HTML output
    contentBuf[0] = '\0';
```

```

strcat(contentBuf, "<HTML>\n");
strcat(contentBuf, "<BODY>\n\n");

strcat(contentBuf, "<P>\n");
strcat(contentBuf, "  <I> cgiprogram1.c V 1.1 Nathan Hetherington </I> \n");
strcat(contentBuf, "</P>\n\n");

strcat(contentBuf, "<P>\n");
strcat(contentBuf, "  <H2> Here's the result: </H2> \n");
strcat(contentBuf, "</P>\n\n");

strcat(contentBuf, "<P>\n");

// for testing only
//sprintf(tmpBuf, "20");
//setenv("CONTENT_LENGTH", tmpBuf, 1);

pName = "CONTENT_LENGTH";
pValue = getenv(pName);



if( pValue )
{
    sprintf(tmpBuf, "  <B> %s = %s </B><BR>\n", pName, pValue);
    strcat(contentBuf, tmpBuf);

    bytesLeft = atoi(pValue);
    do
    {
        bytesRead = read(STDIN, &readBuf[0], READ_BUFLEN);

        // null-terminate
        if( bytesRead > 0 )
        {
            readBuf[bytesRead] = '\0';
            bytesLeft -= bytesRead;
        }
    } while( (bytesLeft > 0) && (bytesRead > 0) );

    if( bytesRead > 0 )
    {
        sprintf(tmpBuf, "  <B> data read: </B>\n");
        strcat(contentBuf, tmpBuf);

        strcat(contentBuf, "   <BR>\n");
        strcat(contentBuf, "   <I>\n");

        sprintf(tmpBuf, "  %s\n", readBuf);
        strcat(contentBuf, tmpBuf);
        strcat(contentBuf, "  </I>\n");
    }
    else
    {
        sprintf(tmpBuf, "  <B> no data read </B> \n");
        strcat(contentBuf, tmpBuf);
    }
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}



// Manipulate the input string
    fprintf(Out,"%s\n",readBuf);
    getVariables(readBuf);
    fprintf(Out,"Time value = %s\n",Time);
    fprintf(Out,"Vertical Scale value = %s\n",Vscale);
    fprintf(Out,"Channel selected are = %s\n",Channel);

    //Setting the Parameters for the Oscilloscope
    iprintf(id,"TIM:RANG %s\n",Time);
    iprintf(id,"%s:RANG %s\n",Channel,Vscale);

// Change settings on Oscilloscope

iprompt(id,"TIM:RANG?\n",%t,actual);
strcat(desired, Time);

if(strcmp(actual,desired)==0){
        iprompt(id,"%s:RANG?\n",%t,Channel,actual);
        strcat(desired, Channel);
        strcat(desired, ":");
        strcat(desired, Vscale);

        if(strcmp(actual,desired)==0){
            strcat(contentBuf, "<P> <H1> Changes Successful !!!!!!</H1></P>\n");
        }else{
           strcat(contentBuf, "<P> <H1> Changes Unsuccessful !!!!!!</H1></P>\n");
        }

}else{
        strcat(contentBuf, "<P> <H1> Changes Unsuccessful !!!!!!</H1></P>\n");
    }



                // trailer
    strcat(contentBuf, "</P>\n\n");
    strcat(contentBuf, "</BODY>\n");
    strcat(contentBuf, "</HTML>\n");


    // now create the output according to the HTTP protocol

    // initial string specifies content text/plain, text/html etc
    printf("Content-type: text/html\n");

    // content length
    contentLen = strlen(contentBuf);
    printf("Content-length: %d\n", contentLen);

    // **** blank line is important here (separates header & body) ****
    printf("\n");

    // now the content itself
    printf("%s", contentBuf);
    // Close files
    fclose(Out);

    iclose(id);

        // For WIN16 programs, call _siclcleanup before exiting to release
    // resources allocated by SICL for this application. This call is a
    // no-op for WIN32 programs.
    //_siclcleanup();

    return 0;
}
/////////////////////////////////////////////////////////////////////////////////////
// Function Name: getVariables(char *Buffer)
//
//
/////////////////////////////////////////////////////////////////////////////////////
void getVariables(char *Buffer){

        int index=0,count=0;


        while(Buffer[index]!= '='){
            index++;
    }
        index++;
        while(Buffer[index] != '&'){
            Channel[count]=Buffer[index];
            index++;
            count++;
        }
        count++;
        index++;
        Channel[count]='\0';
    printf("Time value = %s\n",Channel);
    count=0;
        while(Buffer[index]!= '='){
            index++;
    }
        index++;
        while(Buffer[index] != '&'){
            Time[count]=Buffer[index];
            index++;
            count++;
        }
        count++;
        index++;
        Time[count]='\0';
    printf("Time value = %s\n",Time);

        while(Buffer[index]!= '='){
            index++;
        }
        count = 0;
        index++;
        while(Buffer[index] != '\0'){
            Vscale[count]=Buffer[index];
            index++;
            count++;
        }
        count++;
        Vscale[count]='\0';
    printf("Vertical Scale value = %s\n",Vscale);

}

```

E.2.3 Switch Unit

Open

```
// This is the first draft of the cgi program for the Switch Unit to open switches
/* 34970open.c
 * CGI using POST method (standard input/output)
 * Note that this method is different to the GET method –
 * the input comes from standard input (stdin), and
 * not the environment variable QUERY_STRING
 *
 * See cgipost.html for use in conjunction with a HTML file.


    Here's the result:

    CONTENT_LENGTH = 98
    data read:
    TEXT_LINE_ONE=test&TEXT_LINE_TWO=default+value&TEXT_LINE_THREE=0123&PASSWORD_FIELD=secr
 *
 *
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sicl.h"

#define DEVICE_ADDRESS "hpib7,10"


#define   MAX_CGILEN  2048
#define   TMP_BUFLEN  512
#define   READ_BUFLEN 512

// holds the contents of the http output (not including the header)
static char contentBuf[MAX_CGILEN+1];

#ifndef   STDIN
#define   STDIN   0
#endif

#ifndef   STDOUT
#define   STDOUT 1
#endif

```

```
// Function Prototypes
void getVariables(char *readBuf);

// Variables for changes to equiptment
char chan_list[30];
char actual[30];
char desired[30];

int     main()
{

    int     bytesRead, bytesLeft;
    char    *pName, *pValue;
    int     contentLen;
    char    tmpBuf[TMP_BUFLEN+1];
    char    readBuf[READ_BUFLEN+1];
        INST id;
        char buff[256];

    // Install a default SICL error handler that logs an error message and
        // exits.  On Windows 95 view messages with the SICL Message Viewer,
        // and on Windows NT
        ionerror(I_ERROR_EXIT);

        // Open a device session using the DEVICE_ADDRESS
        id=iopen(DEVICE_ADDRESS);
        // Write the *RST string (and send an EOI indicator) to put the instrument
    // in a known state.
    iprintf(id, "*RST\n");
        ipromptf(id, "*IDN?\n", "%t", buff);
        //printf("Device = %s\n",buff);

    // buffer which contains the content, ie HTML output
    contentBuf[0] = '\0';

    strcat(contentBuf, "<HTML>\n");
    strcat(contentBuf, "<BODY>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "   <I> 34970open.c V 1.1 Nathan Hetherington </I> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");
    strcat(contentBuf, "   <H2> Here's the result: </H2> \n");
    strcat(contentBuf, "</P>\n\n");

    strcat(contentBuf, "<P>\n");

```

```
// for testing only
//sprintf(tmpBuf, "20");
//setenv("CONTENT_LENGTH", tmpBuf, 1);

pName = "CONTENT_LENGTH";
pValue = getenv(pName);



if( pValue )
{
    sprintf(tmpBuf, "  <B> %s = %s </B><BR>\n", pName, pValue);
    strcat(contentBuf, tmpBuf);

    bytesLeft = atoi(pValue);
    do
    {
        bytesRead = read(STDIN, &readBuf[0], READ_BUFLEN);

        // null-terminate
        if( bytesRead > 0 )
        {
            readBuf[bytesRead] = '\0';
            bytesLeft -= bytesRead;
        }
    } while( (bytesLeft > 0) && (bytesRead > 0) );

    if( bytesRead > 0 )
    {
        sprintf(tmpBuf, "  <B> data read: </B>\n");
        strcat(contentBuf, tmpBuf);

        strcat(contentBuf, "  <BR>\n");
        strcat(contentBuf, "  <I>\n");

        sprintf(tmpBuf, "  %s\n", readBuf);
        strcat(contentBuf, tmpBuf);
        strcat(contentBuf, "  </I>\n");
    }
    else
    {
        sprintf(tmpBuf, "  <B> no data read </B> \n");
        strcat(contentBuf, tmpBuf);
    }
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}

// Apply changes on Switch Unit
iprintf(id,"ROUT:OPEN %s\n",chan_list);
iprompt(id,"ROUT:OPEN?\n",%t,actual);

if(strcmp(actual,desired)==0){
    strcat(contentBuf, "<P> <H1> Changes Successful !!!!!!</H1></P>\n");
}else{
    strcat(contentBuf, "<P> <H1> Changes Unsuccessful !!!!!!</H1></P>\n");
}


    // trailer
strcat(contentBuf, "</P>\n\n");
strcat(contentBuf, "</BODY>\n");
strcat(contentBuf, "</HTML>\n");


// now create the output according to the HTTP protocol

// initial string specifies content text/plain, text/html etc
printf("Content-type: text/html\n");

// content length
contentLen = strlen(contentBuf);
printf("Content-length: %d\n", contentLen);

// **** blank line is important here (separates header & body) ****
printf("\n");

// now the content itself
printf("%s", contentBuf);


// Manipulate the input string
    fprintf(Out,"%s\n",readBuf);
    getVariables(readBuf);

    //iclose(id);

    // For WIN16 programs, call _siclcleanup before exiting to release
// resources allocated by SICL for this application. This call is a
// no-op for WIN32 programs.
//_siclcleanup();

return 0;
```

```
}
////////////////////////////////////////////////////////////////////////////////
// Function Name: getVariables(char *Buffer)
//
//
////////////////////////////////////////////////////////////////////////////////
void getVariables(char *Buffer){

        int index=0,count=0;


        while(Buffer[index]!= '='){
                index++;
    }
        index++;
        while(Buffer[index] != '&'){
                chan_list[count]=Buffer[index];
                index++;
                count++;
        }
        chan_list='\0'

}
```

Close

```
// This is the first draft of the cgi program for the switch unit to close switches.
/* 34970close.c
 * CGI using POST method (standard input/output)
 * Note that this method is different to the GET method –
 * the input comes from standard input (stdin), and
 * not the environment variable QUERY_STRING
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sicl.h"

#define DEVICE_ADDRESS "hpib7,9"


#define   MAX_CGILEN  2048
#define   TMP_BUFLEN  512
#define   READ_BUFLEN 512
```

```

// holds the contents of the http output (not including the header)
static char contentBuf[MAX_CGILEN+1];

#ifndef  STDIN
#define  STDIN  0
#endif

#ifndef  STDOUT
#define  STDOUT 1
#endif

// Function Prototypes
void getVariables(char *readBuf);

// Variables for changes to equiptment
char chan_list[30];
char actual[30];
char desired[30];

int      main()
{
        FILE   *Out;
    int      bytesRead, bytesLeft;
    char     *pName, *pValue;
    int      contentLen;
    char     tmpBuf[TMP_BUFLEN+1];
    char     readBuf[READ_BUFLEN+1];
        INST id;
        char buff[256];

    // Install a default SICL error handler that logs an error message and
        // exits.  On Windows 95 view messages with the SICL Message Viewer,
        // and on Windows NT
        ionerror(I_ERROR_EXIT);

        // Open a device session using the DEVICE_ADDRESS
        id=iopen(DEVICE_ADDRESS);
        // Write the *RST string (and send an EOI indicator) to put the instrument
    // in a known state.
    iprintf(id, "*RST\n");
        ipromptf(id, "*IDN?\n", "%t", buff);
        //printf("Device = %s\n",buff);

        // Open the output file
        Out=fopen("output.txt","w");

    // buffer which contains the content, ie HTML output
```

```
contentBuf[0] = '\0';

strcat(contentBuf, "<HTML>\n");
strcat(contentBuf, "<BODY>\n\n");

strcat(contentBuf, "<P>\n");
strcat(contentBuf, "  <I> 34970cloase.c V 1.1 Nathan Hetherington </I> \n");
strcat(contentBuf, "</P>\n\n");

strcat(contentBuf, "<P>\n");
strcat(contentBuf, "  <H2> Here's the result: </H2> \n");
strcat(contentBuf, "</P>\n\n");

strcat(contentBuf, "<P>\n");

// for testing only
//sprintf(tmpBuf, "20");
//setenv("CONTENT_LENGTH", tmpBuf, 1);

pName = "CONTENT_LENGTH";
pValue = getenv(pName);



if( pValue )
{
    sprintf(tmpBuf, "  <B> %s = %s </B><BR>\n", pName, pValue);
    strcat(contentBuf, tmpBuf);

    bytesLeft = atoi(pValue);
    do
    {
        bytesRead = read(STDIN, &readBuf[0], READ_BUFLEN);

        // null-terminate
        if( bytesRead > 0 )
        {
            readBuf[bytesRead] = '\0';
            bytesLeft -= bytesRead;
        }
    } while( (bytesLeft > 0) && (bytesRead > 0) );

    if( bytesRead > 0 )
    {
        sprintf(tmpBuf, "  <B> data read: </B>\n");
        strcat(contentBuf, tmpBuf);

        strcat(contentBuf, "  <BR>\n");
        strcat(contentBuf, "   <I>\n");

        sprintf(tmpBuf, "   %s\n", readBuf);
        strcat(contentBuf, tmpBuf);
        strcat(contentBuf, "   </I>\n");
    }
    else
    {
        sprintf(tmpBuf, "  <B> no data read </B> \n");
        strcat(contentBuf, tmpBuf);
    }
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}

    // trailer
strcat(contentBuf, "</P>\n\n");
strcat(contentBuf, "</BODY>\n");
strcat(contentBuf, "</HTML>\n");


// now create the output according to the HTTP protocol

// initial string specifies content text/plain, text/html etc
printf("Content-type: text/html\n");

// content length
contentLen = strlen(contentBuf);
printf("Content-length: %d\n", contentLen);

// **** blank line is important here (separates header & body) ****
printf("\n");

// now the content itself
printf("%s", contentBuf);


// Manipulate the input string
    fprintf(Out,"%s\n",readBuf);
    getVariables(readBuf);

// Apply changes on Switch Unit
iprintf(id,"ROUT:CLOS %s\n",chan_list);
iprompt(id,"ROUT:CLOS?\n",%t,actual);

```

```
    if(strcmp(actual,desired)==0){
        strcat(contentBuf, "<P> <H1> Changes Successful !!!!!!</H1></P>\n");
    }else{
        strcat(contentBuf, "<P> <H1> Changes Unsuccessful !!!!!!</H1></P>\n");
    }



        //iclose(id);

        // For WIN16 programs, call _siclcleanup before exiting to release
    // resources allocated by SICL for this application. This call is a
    // no-op for WIN32 programs.
    //_siclcleanup();

    return 0;
}
//////////////////////////////////////////////////////////////////////////////////
// Function Name: getVariables(char *Buffer)
//
//
//////////////////////////////////////////////////////////////////////////////////
void getVariables(char *Buffer){

        int index=0,count=0;


        while(Buffer[index]!= '='){
                index++;
    }
        index++;
        while(Buffer[index] != '&'){
                chan_list[count]=Buffer[index];
                index++;
                count++;
        }
        chan_list='\0'

}

```

E.2.4 Power Supply

```
// This is the first draft of the cgi program for the Power Supply
/* 6624a.c
 * CGI using POST method (standard input/output)
 * Note that this method is different to the GET method –
 * the input comes from standard input (stdin), and
 * not the environment variable QUERY_STRING
 *
 *
 *
 * =

   Here's the result:

   CONTENT_LENGTH = 98
   data read:
   TEXT_LINE_ONE=test&TEXT_LINE_TWO=default+value&TEXT_LINE_THREE=0123&PASSWORD_FIELD=secr
 *
 *
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "sicl.h"

#define DEVICE_ADDRESS "hpib7,5"


#define   MAX_CGILEN  2048
#define   TMP_BUFLEN  512
#define   READ_BUFLEN 512

// holds the contents of the http output (not including the header)
static char contentBuf[MAX_CGILEN+1];

#ifndef   STDIN
#define   STDIN   0
#endif

#ifndef   STDOUT
#define   STDOUT 1
#endif

// Function Prototypes
void getVariables(char *readBuf);

// Variables for changes to equiptment
char Channel[5];
char Volts[20];
char desired[30],actual[30];

int     main()
{

   int     bytesRead, bytesLeft;
   char    *pName, *pValue;
   int     contentLen;
   char    tmpBuf[TMP_BUFLEN+1];
   char    readBuf[READ_BUFLEN+1];
       INST id;
       char buff[256];

   // Install a default SICL error handler that logs an error message and
       // exits.  On Windows 95 view messages with the SICL Message Viewer,
       // and on Windows NT
       ionerror(I_ERROR_EXIT);

       // Open a device session using the DEVICE_ADDRESS
       id=iopen(DEVICE_ADDRESS);

       // Write the *RST string (and send an EOI indicator) to put the instrument
       // in a known state.
       iprintf(id, "*RST\n");
       ipromptf(id, "*IDN?\n", "%t", buff);
       printf("Device = %s\n",buff);


   // buffer which contains the content, ie HTML output
   contentBuf[0] = '\0';

   strcat(contentBuf, "<HTML>\n");
   strcat(contentBuf, "<BODY>\n\n");

   strcat(contentBuf, "<P>\n");
   strcat(contentBuf, "  <I> 6624A.c V 1.1 Nathan Hetherington </I> \n");
   strcat(contentBuf, "</P>\n\n");

   strcat(contentBuf, "<P>\n");
   strcat(contentBuf, "  <H2> Here's the result: </H2> \n");
   strcat(contentBuf, "</P>\n\n");

   strcat(contentBuf, "<P>\n");

   // for testing only
   //sprintf(tmpBuf, "20");
   //setenv("CONTENT_LENGTH", tmpBuf, 1);

```

```
pName = "CONTENT_LENGTH";
pValue = getenv(pName);



if( pValue )
{
    sprintf(tmpBuf, "  <B> %s = %s </B><BR>\n", pName, pValue);
    strcat(contentBuf, tmpBuf);

    bytesLeft = atoi(pValue);
    do
    {
        bytesRead = read(STDIN, &readBuf[0], READ_BUFLEN);

        // null-terminate
        if( bytesRead > 0 )
        {
            readBuf[bytesRead] = '\0';
            bytesLeft -= bytesRead;
        }
    } while( (bytesLeft > 0) && (bytesRead > 0) );

    if( bytesRead > 0 )
    {
        sprintf(tmpBuf, "  <B> data read: </B>\n");
        strcat(contentBuf, tmpBuf);

        strcat(contentBuf, "   <BR>\n");
        strcat(contentBuf, "   <I>\n");

        sprintf(tmpBuf, "  %s\n", readBuf);
        strcat(contentBuf, tmpBuf);
        strcat(contentBuf, "   </I>\n");
    }
    else
    {
        sprintf(tmpBuf, "  <B> no data read </B> \n");
        strcat(contentBuf, tmpBuf);
    }
}
else
{
    sprintf(tmpBuf, "  <B> %s is undefined </B> \n", pName);
    strcat(contentBuf, tmpBuf);
}

    // Manipulate the input string
```

```
        fprintf(Out,"%s\n",readBuf);
        getVariables(readBuf);

        // Set the Power Supply
        iprintf(id,"VOLT %s,%s, OV\n",Channel,Volts);

        // trailer
        strcat(contentBuf, "</P>\n\n");
        strcat(contentBuf, "</BODY>\n");
        strcat(contentBuf, "</HTML>\n");


        // now create the output according to the HTTP protocol

        // initial string specifies content text/plain, text/html etc
        printf("Content-type: text/html\n");

        // content length
        contentLen = strlen(contentBuf);
        printf("Content-length: %d\n", contentLen);

        // **** blank line is important here (separates header & body) ****
        printf("\n");

        // now the content itself
        printf("%s", contentBuf);

        // Close files
        fclose(Out);
        iclose(id);

        // For WIN16 programs, call _siclcleanup before exiting to release
        // resources allocated by SICL for this application. This call is a
        // no-op for WIN32 programs.
        _siclcleanup();

        return 0;
}
////////////////////////////////////////////////////////////////////////////////
// Function Name: getVariables(char *Buffer)
//
//
////////////////////////////////////////////////////////////////////////////////
void getVariables(char *Buffer){

        int index=0,numVar=1,count=0;

        while(Buffer[index]!= '='){
                index++;
        }
        index++;
        while(Buffer[index] != '&'){
                Channel[count]=Buffer[index];
                index++;
                count++;
        }
        count++;
        Frequency[count]='\0';

        while(Buffer[index]!= '='){
                index++;
        }


        count = 0;
        index++;
        while(Buffer[index] != '\0'){
                Volts[count]=Buffer[index];
                index++;
                count++;
        }

        Volts[count]='\0';
}
```

Appendix F

Basic C Programs

F.1 Data Types

```
/**********************************************************
* File Name: datatypes.c                                   *
*                                                          *
* Description: This program defines various data types and *
*              prints their size in bytes.                 *
*                                                          *
* Written By: Nathan Hetherington                          *
**********************************************************/
#include <stdio.h>
#include <stdlib.h>


int main(void){

    char Char;
    int integer;
    long int l_int;
    float Float;
    long Long;
    double Double;

    printf("The size of an int is %d\n",sizeof(integer));
    printf("The size of an char is %d\n",sizeof(Char));
    printf("The size of an float is %d\n",sizeof(Float));
    printf("The size of an long is %d\n",sizeof(Long));
    printf("The size of an double is %d\n",sizeof(Double));
    printf("The size of an long int is %d\n",sizeof(l_int));

    sleep(5000);
    return 0;

}


```

F.2 Structures

```
/*********************************************************
* File Name: struct.c                                     *
*                                                         *
* Description: This program defines a stucture data types *
*              and prints their contents.                 *
*                                                         *
* Written By: Nathan Hetherington                         *
*********************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void){

        typedef struct Student{
                int age, student_number;
                char name[30];
            float GPA;
        }STUDENT;

        STUDENT St1;

        St1.age=21;
        St1.student_number=22556677;
        strcpy(St1.name,"Nathan Hetherington");
        St1.GPA=4.76;


        printf("\nThe student %s of age %d \n",St1.name,St1.age);
        printf("has the student number %d and GPA of %f\n",St1.student_number,St1.GPA);


        sleep(5000);
        return 0;
}
```

F.3 Variable Scope

```c
/*******************************************************
* File Name: scope.c                                   *
*                                                      *
* Description: This program demonstrates the scope of  *
*              variables in C                          *
*                                                      *
* Written By: Nathan Hetherington                      *
*******************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int x;

// Defines sub routine
void sub1(void){
        int x=50;
        printf("x is %d in Sub1\n",x);
}

int main(void){

        x=10;
        printf("x is %d in the main function \n",x);

        sub1();

        printf("x is %d after sub1\n",x);

        sleep(5000);
        return 0;
}
```

F.4 Input/Output Example

```c
/*******************************************************
* File Name: inout.c                                   *
*                                                      *
* Description: This program demonstrates the basic input and*
*              output.                                 *
*                                                      *
* Written By: Nathan Hetherington                    *
*******************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int main(void){

    int integer;
    char character,string[20];


    /* This is the Input Phase */
    printf("\nEnter A integer : ");
    scanf("%d",&integer);

    printf("\nEnter a Character : ");
    scanf("%c",&character);

    printf("\nEnter a string(Enter terminates input) : ");
    gets(string);

    /* This is the Output phase */
    printf("\nThe Integer entered is %d",integer);
    printf("\nThe Character entered is %c",character);
    printf("\n%s is the string\n",string);

    sleep(5000);
    return 0;

}
```

Appendix G

Application Interface Library

G.1 VISA Library

Agilent VISA User's Guide

Agilent Technologies

Manual Part Number: E2090-90040
Printed in U.S.A. E0701

Contents
Agilent VISA User's Guide

Front Matter **9**
- Notice 9
- Warranty Information 9
- U.S. Government Restricted Rights 9
- Trademark Information 10
- Printing History 10
- Copyright Information 10

1. Introduction **11**
- What's in This Guide? 13
- VISA Overview 14
  - Using VISA and SICL 14
  - VISA Support 15
  - VISA Documentation 16
  - Contacting Agilent 16

2. Building a VISA Application in Windows **17**
- Building a VISA Program (C/C++) 19
  - Compiling and Linking VISA Programs (C/C++) 19
  - Example VISA Program (C/C++) 21
- Building a VISA Program (Visual Basic) 23
  - Visual Basic Programming Considerations 23
  - Example VISA Program (Visual Basic) 25
- Logging Error Messages 29
  - Using the Event Viewer 29
  - Using the Message Viewer 29
  - Using the Debug Window 30

3. Building a VISA Application in HP-UX **31**
- Building a VISA Program in HP-UX 33
  - Example Source Code 33
  - Example Program Contents 34
  - Running the Example Program 34
  - Compiling and Linking a VISA Program 35
  - Logging Error Messages 35
- Using Online Help 36
  - Using the HyperHelp Viewer 36
  - Using HP-UX Manual Pages 36

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

viLock 213
viMapAddress 217
viMapTrigger 220
viMemAlloc 223
viMemFree 225
viMove 226
viMoveAsync 229
viMoveIn8, viMoveIn16, and viMoveIn32 233
viMoveOut8, viMoveOut16, and viMoveOut32 236
viOpen 239
viOpenDefaultRM 243
viOut8, viOut16, and viOut32 245
viParseRsrc 248
viPeek8, viPeek16, and viPeek32 251
viPoke8, viPoke16, and viPoke32 252
viPrintf 253
viQueryf 262
viRead 264
viReadAsync 267
viReadSTB 269
viReadToFile 271
viScanf 274
viSetAttribute 284
viSetBuf 286
viSPrintf 288
viSScanf 290
viStatusDesc 292
viTerminate 293
viUninstallHandler 295
viUnlock 297
viUnmapAddress 298
viUnmapTrigger 299
viVPrintf 301
viVQueryf 303
viVScanf 305
viVSPrintf 307
viVSScanf 309
viVxiCommandQuery 311
viWaitOnEvent 314
viWrite 320
viWriteAsync 322
viWriteFromFile 324

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

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

Trademark Information

Printing History

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

Copyright Information

1

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. F*or 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 VXI*plug&play* System Alliance that allows software developed from different vendors to run on the same system.

Use VISA if you want to use VXI*plug&play* instrument drivers in your applications, or if you want the I/O applications or instrument drivers that you develop to be compliant with VXI*plug&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 VXI*plug&play* standards. Applications and instrument drivers developed with VISA can execute on VXI*plug&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 VXI*plug&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 VXI*plug&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 Documentation

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

Agilent VISA Documentation

Document	Description
Agilent IO Libraries Installation and Configuration Guide for Windows	Shows how to install, configure, and maintain the Agilent IO Libraries on Windows.
Agilent IO Libraries Installation and Configuration Guide for HP-UX	Shows how to install, configure, and maintain the Agilent IO Libraries on HP-UX.
VISA Online Help	Information is provided in the form of Windows Help.
VISA Example Programs	Example programs are provided online to help you develop VISA applications.
VXI*plug&play* System Alliance *VISA Library Specification 4.3*	Specifications for VISA.
IEEE Standard Codes, Formats, Protocols, and Common Commands	ANSI/IEEE Standard 488.2-1992.
VXIbus Consortium specifications (when using VISA over LAN)	*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

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`

2

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.

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

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

/* Close session */
viClose(vi);
viClose(defaultRM);}
```

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.

Function(s)	Description
`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.

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`**, **`viScanf`**, **`viQueryf`**) are not callable from VB. Use the corresponding **`viVPrintf`**, **`viVScanf`** 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'.

Format Conversion Commands

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:

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

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

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

Numeric Arrays

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.

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

Strings

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:

```
Dim strVal as String
```

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

```
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.

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.

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.

7 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`**.

8 The name and location of the VB declaration file depends on which operating system is used. Assuming the 'standard' VISA directory C:\Program Files\Visa or the 'standard' VXI*pnp* directory C:\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)

9 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.

10 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.

```
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 "GPIB0" 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.
     ' "GPIB0::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, "GPIB0::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.

Function(s)	Description
`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 (*defrm*) to that resource manager session.
`viOpen`	This function establishes a communication channel with the device specified. A session identifier (**`vi`**) that can be used with other VISA functions is returned. This call must be made for each device you will be using.
`viVPrintf` and **`viVScanf`**	These are the VISA formatted I/O functions. The **`viVPrintf`** call sends the IEEE 488.2 **`*RST`** command to the instrument (plus a linefeed character) and puts it in a known state. The **`viVPrintf`** call is used again to query for the device identification (**`*IDN?`**). The **`viVScanf`** call is then used to read the results (*strRes*) that are displayed in a Message Box.
`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.

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.

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.

3

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:

```
idn
```

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

```
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

`-k` *keyword*	Opens the Viewer and searches for the specified *keyword*.
`-p` *partial_keyword*	Opens the Viewer and searches for a specific *partial keyword.*
`-s` *viewmode*	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).
`-display` *display*	Opens the Viewer on the specified *display.*

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
```

4

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

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.

Resource Class	Interface Types	Resource Class Description
Instrument Control (INSTR)	Generic, GPIB, GPIB-VXI, Serial, TCPIP, VXI	Device operations (reading, writing, triggering, etc.).
GPIB Bus Interface (INTFC)	Generic, GPIB	Raw GPIB interface operations (reading, writing, triggering, etc.).
Memory Access (MEMACC)	Generic, GPIB-VXI, VXI	Address space of a memory-mapped bus such as the VXIbus.

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

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.

```
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:

```
#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:

```
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.

Parameter	Description
sesn	A session returned from the **viOpenDefaultRM** function that identifies the resource manager session.
rsrcName	A unique symbolic name of the resource (resource address).

Parameter	Description
accessMode	Specifies the modes by which the resource is to be accessed. The value **`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 **`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 "bit-wise OR" of the values. (Must use **`VI_NULL`** in VISA 1.0.).
timeout	If the *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 **`VI_NULL`** in VISA 1.0.)
vi	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.

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 *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 *scanner* when using other VISA functions. See "Addressing a Session" for information on addressing particular devices.

```
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);
```

Addressing a Session

As shown in the previous section, the *rsrcName* parameter in the **`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 *rsrcName* for different interface types. **`INSTR`** is an optional parameter that indicates that you are communicating with a resource that is of type **`INSTR`**, meaning instrument. The keywords are:

- **`ASRL`** establishes communication with asynchronous serial devices.
- **`GPIB`** establishes communication with GPIB devices or interfaces.
- **`GPIB-VXI`** is used for GPIB-VXI controllers.
- **`TCPIP`** establishes communication with LAN instruments.
- **`VXI`** is used for VXI instruments.

Interface	Typical Syntax
`ASRL`	**`ASRL[`***board***`][::INSTR]`**
`GPIB`	**`GPIB[`***board***`]::`***primary address***`[::`***secondary address***`][::INSTR]`**
`GPIB`	**`GPIB[`***board***`]::INTFC`**
`GPIB-VXI`	**`GPIB-VXI[`***board***`]::`***VXI logical address***`[::INSTR]`**
`GPIB-VXI`	**`GPIB-VXI[`***board***`]::MEMACC`**
`GPIB-VXI`	**`GPIB-VXI[`***board***`][::`***VXI logical address***`]::BACKPLANE`**
`TCPIP`	**`TCPIP[`***board***`]::`***host address***`[::`***LAN device name***`]::INSTR`**
`TCPIP`	**`TCPIP[`***board***`]::`***host address***`::`***port***`::SOCKET`**
`VXI`	**`VXI[`***board***`]::`***VXI logical address***`[::INSTR]`**
`VXI`	**`VXI[`***board***`]::MEMACC`**
`VXI`	**`VXI[`***board***`][::`***VXI logical address***`]::BACKPLANE`**

The following table describes the parameters used above.

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

Some examples of valid symbolic names follow.

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

VXI::1::BACKPLANE	Mainframe resource for chassis 1 on the default VXI system, which is interface 0.
GPIB-VXI2:: BACKPLANE	Mainframe resource for default chassis on GPIB-VXI interface 2.
GPIB1::SERVANT	Servant/device-side resource for GPIB interface 1.
VXI0::SERVANT	Servant/device-side resource for VXI interface 0.
TCPIP0::1.2.3.4::999:: SOCKET	Raw TCPIP access to port 999 at the specified address.
TCPIP::devicename@ company.com::INSTR	TCPIP device using VXI-11 located at the specified address. This uses the default LAN Device Name of *inst0*.

Example: Opening a Session

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

```
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:

```
viFindRsrc(sesn, expr, findList, retcnt, instrDesc);
.
.
viFindNext(findList, instrDesc);
.
.
viClose (findList);
```

Where the parameters are defined as follows.

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

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.

Interface	*expr* Parameter
GPIB	*GPIB[0-9]*::?*INSTR*
VXI	*VXI?*INSTR*
GPIB-VXI	*GPIB-VXI?*INSTR*
GPIB and GPIB-VXI	*GPIB?*INSTR*
All VXI	*?*VXI[0-9]*::?*INSTR*
ASRL	*ASRL[0-9]*::?*INSTR*
All	*?*INSTR*

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.

```
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.

Formatted I/O Functions

As noted, the VISA formatted I/O functions are **`viPrintf`**, **`viQueryf`**, and **`viScanf`.**

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

  ```
  viPrintf(vi, writeFmt[, arg1][, arg2][, ...]);
  viVPrintf(vi, writeFmt, params);
  ```

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

  ```
  viScanf(vi, readFmt[, arg1][, arg2][, ...]);
  viVScanf(vi, readFmt, params);
  ```

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

  ```
  viQueryf(vi, writeFmt, readFmt[, arg1][, arg2][, ...]);
  viVQueryf(vi, writeFmt, readFmt, params);
  ```

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 `%` (percent) followed by an optional modifier(s), followed by a format code.

`%`*[modifiers]format code*

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*:

```
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.

Format Code	Description
d	Indicates the minimum number of digits to appear is specified for the **@1, @H, @Q,** and **@B** flags, and the **i, o, u, x,** and **X** format codes.
f	Indicates the maximum number of digits after the decimal point is specified.
s	Indicates the maximum number of characters for the string is specified.
g	Indicates the maximum significant digits are specified.

Example: Using the Precision Modifier

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*:

```
float numb = 26.9345;
viPrintf(vi, "%.2f\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.

Argument Length Modifier	Format Codes	Description
h	**d, b, B**	Corresponding argument is a short integer or a reference to a short integer for **d**. For **b** or **B**, the argument is the location of a block of data or a reference to a data array. (**B** is only used with **viPrintf** or **viQueryf** (writeFmt).)
l	**d, f, b, B**	Corresponding argument is a long integer or a reference to a long integer for **d**. For **f**, the argument is a double float or a reference to a double float. For b or B, the argument is the location of a block of data or a reference to a data array. (**B** is only used with **viPrintf** or **viQueryf** (writeFmt).)

Argument Length Modifier	Format Codes	Description
L	**f**	Corresponding argument is a long double or a reference to a long double.
z	**b, B**	Corresponding argument is an array of floats or a reference to an array of floats. (**B** is only used with **viPrintf** or **viQueryf** (writeFmt).)
Z	**b, B**	Corresponding argument is an array of double floats or a reference to an array of double floats. (**B** is only used with **viPrintf** or **viQueryf** (writeFmt).)

- **, 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 *dd* is the number of elements to read or write.

  For **`viPrintf`** or **`viQueryf`** (*writeFmt*), you can use an asterisk (`*`) in place of the integer to indicate that the integer is taken from the next argument. For **`viScanf`** or **`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 *vi*:

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

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 **`viPrintf`** and **`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.

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.

Special Characters and Operators	Description
`?`	Matches any one character.
`\`	Makes the character that follows it an ordinary character instead of special character. For example, when a question mark follows a backslash (e.g.,' '\?'), it matches the '?' character instead of any one character.
`[`*`list`*`]`	Matches any one character from the enclosed *list*. A hyphen can be used to match a range of characters.
`[^`*`list`*`]`	Matches any character not in the enclosed *list*. A hyphen can be used to match a range of characters.
`*`	Matches 0 or more occurrences of the preceding character or expression.
`+`	Matches 1 or more occurrences of the preceding character or expression.
`exp|exp`	Matches either the preceding or following expression. The or operator **`|`** matches the entire expression that precedes or follows it and not just the character that precedes or follows it. For example, **`VXI|GPIB`** means **`(VXI) | (GPIB)`**, not **`VXI(I|G)PIB`**.

Special Characters and Operators	Description
`(`*`exp`*`)`	Grouping characters or expressions.
`" "`	Sends a blank space.
`\n`	Sends the ASCII line feed character. The END identifier will also be sent.
`\r`	Sends an ASCII carriage return character.
`\t`	Sends an ASCII TAB character.
`\###`	Sends ASCII character specified by octal value.
`\"`	Sends the ASCII double quote character.
`\\`	Sends a backslash character.

Example Expression	Sample Matches
`GPIB?*INSTR`	Matches `GPIB0::2::INSTR`, `GPIB1::1::1::INSTR`, and `GPIB-VXI1::8::INSTR`
`GPIB[0-9]*::?*INSTR`	Matches `GPIB0::2::INSTR` and `GPIB1::1::1::INSTR` but not `GPIB-VXI1::8::INSTR`
`GPIB[0-9]::?*INSTR`	Matches `GPIB0::2::INSTR` and `GPIB1::1::1::INSTR` but not `GPIB12::8::INSTR`.
`GPIB[^0]::?*INSTR`	Matches `GPIB1::1::1::INSTR` but not `GPIB0::2::INSTR` or `GPIB12::8::INSTR`
`VXI?*INSTR`	Matches `VXI0::1::INSTR` but not `GPIB-VXI0::1::INSTR`
`GPIB-VXI?*INSTR`	Matches `GPIB-VXI0::1::INSTR` but not `VXI0::1::INSTR`
`?*VXI[0-9]*::?*INSTR`	Matches `VXI0::1::INSTR` and `GPIB-VXI0::1::INSTR`
`ASRL[0-9]*::?*INSTR`	Matches `ASRL1::INSTR` but not `VXI0::5::INSTR`
`ASRL1+::INSTR`	Matches `ASRL1::INSTR` and `ASRL11::INSTR` but not `ASRL2::INSTR`

Example Expression	Sample Matches
`(GPIB|VXI)?*INSTR`	Matches `GPIB1::5::INSTR` and `VXI0::3::INSTR` but not `ASRL2::INSTR`
`(GPIB0|VXI0)::1::INSTR`	Matches `GPIB0::1::INSTR` and `VXI0::1::INSTR`
`?*INSTR`	Matches all `INSTR` (device) resources
`?*VXI[0-9]*::?*MEMACC`	Matches `VXI0::MEMACC` and `GPIB-VXI1::MEMACC`
`VXI0::?*`	Matches `VXI0::1::INSTR`, `VXI0::2::INSTR`, and `VXI0::MEMACC`
`?*`	Matches all resources

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

Format Codes	Description
viPrintf/viVPrintf and viQueryf/viVqueryf (*writeFmt*)	
d, i	Corresponding argument is an integer.
f	Corresponding argument is a double.
c	Corresponding argument is a character.
s	Corresponding argument is a pointer to a null terminated string.
%	Sends an ASCII percent (**%**) character.
o, u, x, X	Corresponding argument is an unsigned integer.
e, E, g, G	Corresponding argument is a double.
n	Corresponding argument is a pointer to an integer.
b, B	Corresponding argument is the location of a block of data.
viPrintf/viVPrintf and viQueryf/viVqueryf (*readFmt*)	
d,i,n	Corresponding argument must be a pointer to an integer.
e,f,g	Corresponding argument must be a pointer to a float.
c	Corresponding argument is a pointer to a character sequence.
s,t,T	Corresponding argument is a pointer to a string.
o,u,x	Corresponding argument must be a pointer to an unsigned integer.

Format Codes	Description
[	Corresponding argument must be a character pointer.
b	Corresponding argument is a pointer to a data array.

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`** (*writeFmt*) 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`** (*readFmt*) 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.

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.

```
/*formatio.c
  This example program makes a multimeter measurement
  with a comma-separated list passed with formatted
  I/O and prints the results. You may need to change
  the device address. */

#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(&efaultRM);
  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);
/* Read results */
viScanf(vi, "%lf", &res);

/* Print results */
printf("Measurement Results: %lf\n", res);
/* Close session */
viClose(vi);
viClose(defaultRM);}
```

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.

  `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.

  `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.

  `viReadAsync(`*vi, buf, count, jobId*`);`

- **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 anI/O completion event to identify which asynchronous write operation completed.

  `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.

```
/*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..

Instrument Control (INSTR) Resource Events

`VI_EVENT_SERVICE_REQUEST`
Notification that a service request was received from the device.

Event Attribute	Description	AP	Data Type	Range
`VI_ATTR_EVENT_TYPE`	Unique logical identifier of the event.	RO	**`ViEventType`**	**`VI_EVENT_ SERVICE_REQ`**

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

Event Attributes	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_ VXI_STOP
VI_ATTR_SIGP_ STATUS_ID	The 16-bit Status/ID value retrieved during the IACK cycle or from the Signal register.	RO	ViUInt16	0 to $FFFF_h$

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).

Event Attributes	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_TRIG
VI_ATTR_RECV_TRIG_ ID	The identifier of the triggering mechanism on which the specified trigger event was received.	RO	ViInt16	VI_TRIG_TTL0 to VI_TRIG_TTL7; VI_TRIG_ECL0 to VI_TRIG_ECL1*

* 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.

Event Attributes	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_IO_ COMPLETION
VI_ATTR_STATUS	Return code of the asynch-ronous I/O operation that has completed	RO	ViStatus	N/A
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed	RO	ViJobId	N/A
VI_ATTR_BUFFER	Address of a buffer that was used in an asynchronous operation.	RO	ViBuf	N/A

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

Event Attributes	Description	AP	Data Type	Range
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to $FFFFFFFF_h$
VI_ATTR_OPER_NAME	Name of the operation generating the event.		ViString	N/A

Memory Access (MEMACC) Resource Event

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_ IO_COMPLETION
VI_ATTR_STATUS	Return code of the asynch-ronous I/O operation that has completed.	RO	ViStatus	N/A
VI_ATTR_JOB_ID	Job ID of the asynchronous operation that has completed.	RO	ViJobId	N/A
VI_ATTR_BUFFER	Address of a buffer that was used in an asynchronous operation.	RO	ViBuf	N/A
VI_ATTR_RET_COUNT	Actual number of elements that were asynchronously transferred.	RO	ViUInt32	0 to $FFFFFFFF_h$
VI_ATTR_OPER_NAME	Name of the operation generating the event.	RO	ViString	N/A

.

GPIB Bus Interface (INTFC) Resource Events

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

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_GPIB_ CIC
VI_ATTR_GPIB_RECV_ CIC_STATE	Controller has become controller-in-charge.	RO	ViBoolean	VI_TRUE VI_FALSE

VI_EVENT_GPIB_TALK
Notification that the GPIB controller has been addressed to talk

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_GPIB_ TALK

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

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_GPIB_ LISTEN

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

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	ViEventType	VI_EVENT_CLEAR

`VI_EVENT_TRIGGER`
Notification that a trigger interrupt was received from the interface.

Event Attribute	Description	AP	Data Type	Range
`VI_ATTR_EVENT_TYPE`	Unique logical identifier of the event.	RO	**`ViEventType`**	**`VI_EVENT_TRIG`**
`VI_ATTR_RECV_TRIG_ ID`	The identifier of the triggering mechanism on which the specified trigger event was received.	RO	**`ViInt16`**	**`VI_TRIG_SW`**

`VI_EVENT_IO_COMPLETION`
Notification that an asynchronous operation has completed.

Event Attribute	Description	AP	Data Type	Range
`VI_ATTR_EVENT_TYPE`	Unique logical identifier of the event.	RO	**`ViEventType`**	**`VI_EVENT_IO_ COMPLETION`**
`VI_ATTR_STATUS`	Return code of the asynch-ronous I/O operation that has completed.	RO	**`ViStatus`**	`N/A`
`VI_ATTR_JOB_ID`	Job ID of the asynchronous operation that has completed.	RO	**`ViJobId`**	`N/A`
`VI_ATTR_BUFFER`	Address of buffer used in an asynchronous operation.	RO	**`ViBuf`**	`N/A`
`VI_ATTR_RET_COUNT`	Actual number of elements that were asynchronously transferred.	RO	**`ViUInt32`**	0 to $\mathrm{FFFFFFFF}_h$
`VI_ATTR_OPER_NAME`	The name of the operation generating the event.	RO	**`ViString`**	`N/A`

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).

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	**ViEventType**	**VI_EVENT_TRIG**
VI_ATTR_RECV_TRIG_ID	The identifier of the triggering mechanism on which the specified trigger event was received.	RO	**ViInt16**	**VI_TRIG_TTL0** to **VI_TRIG_TTL7**; **VI_TRIG_ECL0** to **VI_TRIG_ECL1**

VI_EVENT_VXI_VME_SYSFAIL
Notification that the VXI/VME SYSFAIL* line has been asserted.

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	**ViEventType**	**VI_EVENT_VXI_VME_SYSFAIL**

VI_EVENT_VXI_VME_SYSRESET
Notification that the VXI/VME SYSRESET* line has been reset

Event Attribute	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	**ViEventType**	**VI_EVENT_VXI_VME_SYSRESET**

TCPIP Socket (SOCKET) Resource Event

VI_EVENT_IO_COMPLETION
Notification that an asynchronous operation has completed.

Event Attributes	Description	AP	Data Type	Range
VI_ATTR_EVENT_TYPE	Unique logical identifier of the event.	RO	**ViEventType**	**VI_EVENT_IO_COMPLETION**
VI_ATTR_STATUS	Return code of the asynchronous I/O operation that has completed	RO	**ViStatus**	N/A

TCPIP Socket (SOCKET) Resource Event

`VI_EVENT_IO_COMPLETION`
Notification that an asynchronous operation has completed.

Event Attributes	Description	AP	Data Type	Range
`VI_ATTR_JOB_ID`	Job ID of the asynchronous operation that has completed	RO	**`ViJobId`**	`N/A`
`VI_ATTR_BUFFER`	Address of a buffer that was used in an asynchronous operation.	RO	**`ViBuf`**	`N/A`
`VI_ATTR_RET_COUNT`	Actual number of elements that were asynchronously transferred.	RO	**`ViUInt32`**	`0 to` $\texttt{FFFFFFFF}_h$
`VI_ATTR_OPER_NAME`	Name of the operation generating the event.	RO	**`ViString`**	`N/A`

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.

```
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

Example: Using the Callback Method

This example shows one way you can use the callback method.

```
ViStatus _VI_FUNCH 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 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:

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

Where the parameters are defined as follows:

Parameter	Description
vi	The session on which the handler will be installed.
eventType	The event type that will activate the handler.
handler	The name of the handler to be called.
userHandle	A user value that uniquely identifies the handler for the specified event type.

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.

```
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.

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:

```
viEnableEvent(vi, eventType, mechanism, 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.

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

Example: Enabling a Hardware Trigger Event

This example illustrates enabling a hardware trigger event.

```
viInstallHandler(vi, VI_EVENT_TRIG, my_handler,&addr);
viEnableEvent(vi, VI_EVENT_TRIG, VI_HNDLR, VI_NULL);
```

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, 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");

   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 uninstall 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 )
#elif 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;
```

```
}
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 ." );
  for (count = 0 ; (count < 10) && (srqOccurred ==
  0);count++) {
    long count2 = 0;
    printf( "." );
    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,
(ViAddr)10);
/* Clean up - do not leave device in error state */
viPrintf( vi, "*CLS\n" );

/* close the sessions */
viClose(vi);
viClose(defaultRM);
printf( "End of program\n" );}
```

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.

```
   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_INFINITE,
     &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:

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

where the parameters are defined as follows:

Parameter	Description
vi	The session the handler will be installed on.
eventType	The type of event to enable.
mechanism	The mechanism by which the event will be enabled. Specify **`VI_QUEUE`** to use the queuing method.
context	Not used in VISA 1.0. Use **`VI_NULL`**.

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.

```
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:

```
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.

```
  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.

```
/* 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);
}
```

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:

```
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.

```
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:

```
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.

```
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.

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 applicationfrom 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.

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.

Attribute Name	Access Privilege		Data Type	Range	Default
`VI_ATTR_EVENT_TYPE`	RO	Global	**`ViEventType`**	**`VI_EVENT_EXCEPTION`**	N/A
`VI_ATTR_STATUS`	RO	Global	**`ViStatus`**	N/A	N/A
`VI_ATTR_OPER_NAME`	RO	Global	**`ViString`**	N/A	N/A

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.*/

#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.

`viLock(`*vi, lockType, timeout, requestedKey, 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 atttributes.

- **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.

Example: Exclusive Lock

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\n");

   /* 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?\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);
   /* Close session */
   viClose (vi);
   viClose (defaultRM);}
```

Example: Shared Lock

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);
}
```

Notes:

5

Programming via GPIB and VXI

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).

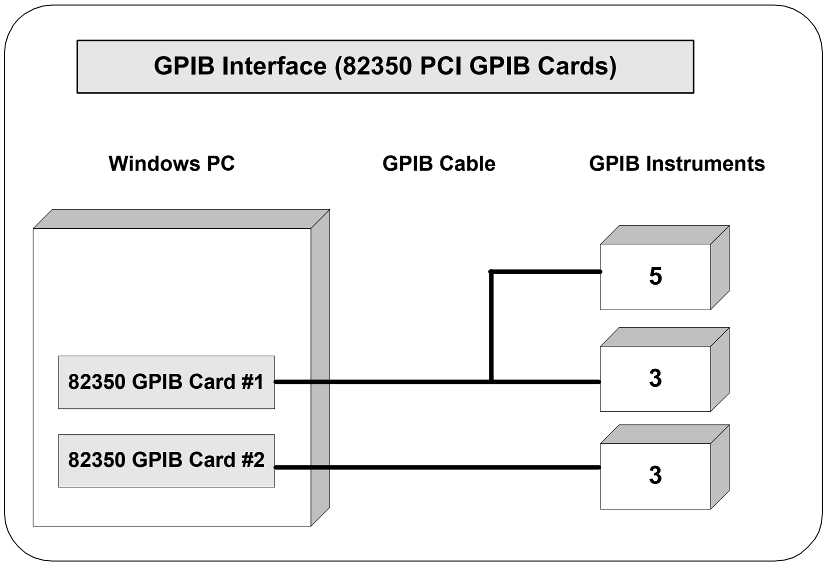
A **message-based VXI device** has its own processor that allows it to interpret high-level commands such as Standard Commands for Programmable Instruments (SCPI). When using VISA, you can place the SCPI command within your VISA output function call. Then, the message-based device interprets the SCPI 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.

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.

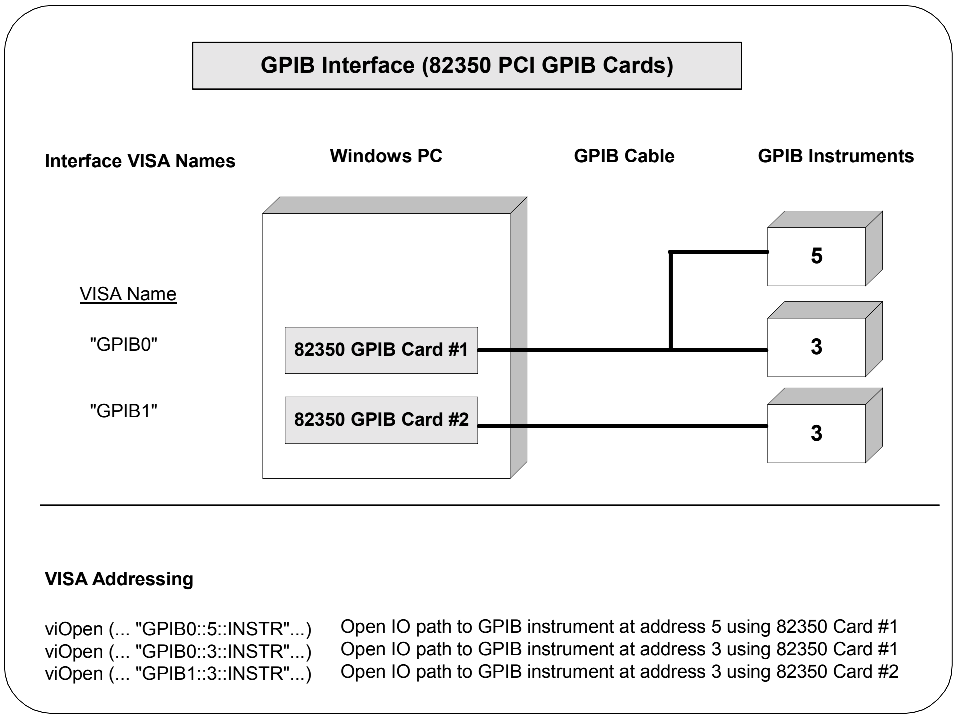
GPIB Interfaces Overview

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

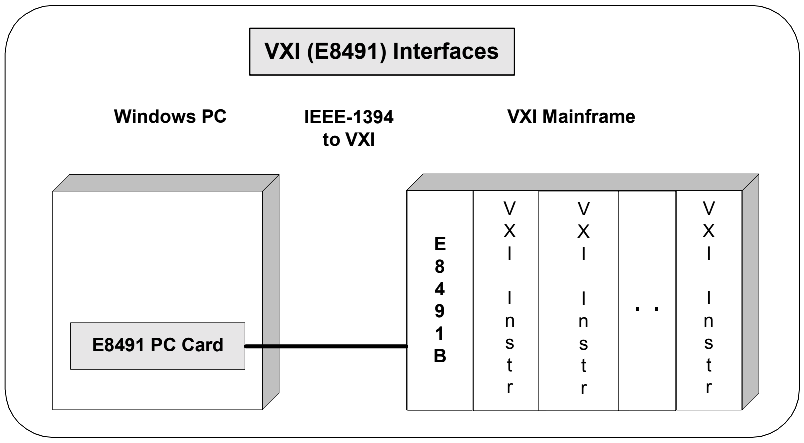
The GPIB interface system in the following figure consists of a Windows PC with two 82350 GPIB cards connected to three GPIB instruments via GPIB cables. For this system, the IO Config utility has been used to assign GPIB card #1 a VISA name of “GPIB0” and to assign GPIB card #2 a VISA name of “GPIB1”. VISA addressing is as shown in the figure.



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

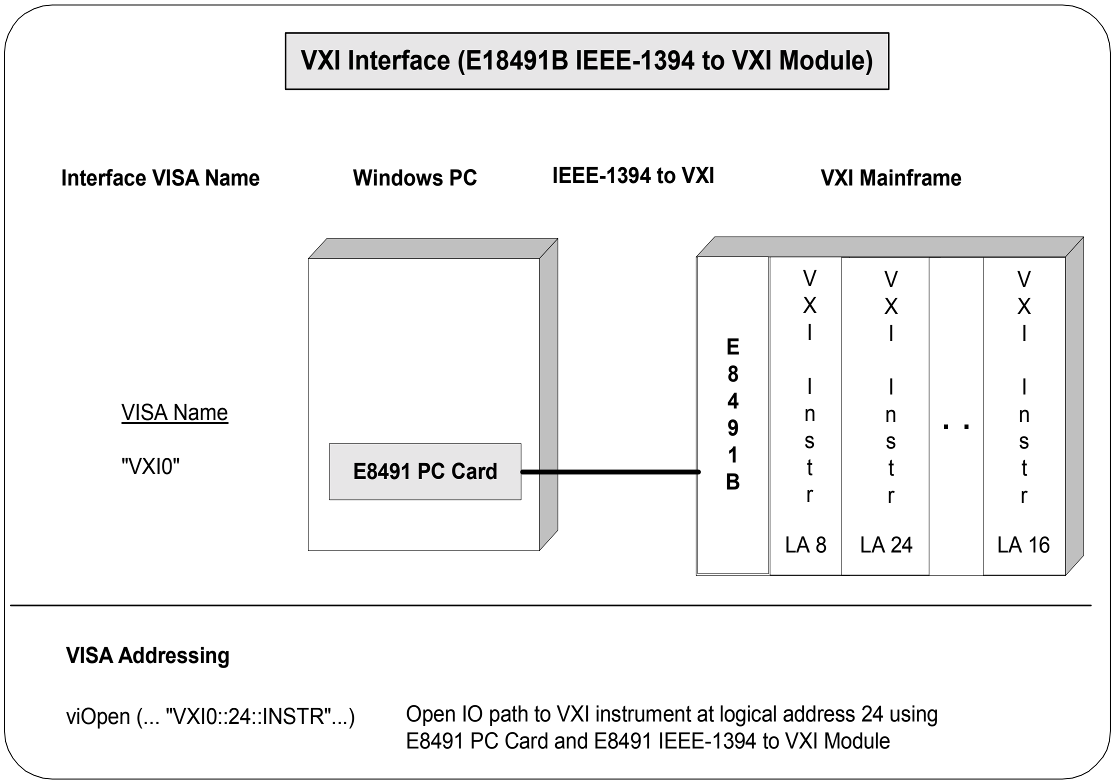
As shown in the following figure, a typical VXI (E8491) interface consists of an E8491 PC Card in a Windows PC that is connected to an E8491B IEEE-1394 Module in a VXI mainframe via an IEEE-1394 to VXI cable. The VXI mainframe also includes one or more VXI instruments.



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.*



VISA Addressing

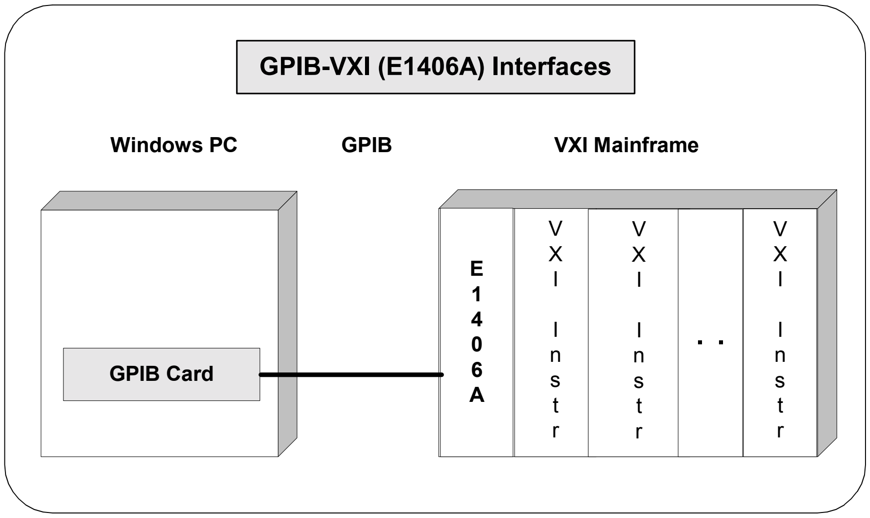
viOpen (... "VXI0::24::INSTR"...)	Open IO path to VXI instrument at logical address 24 using E8491 PC Card and E8491 IEEE-1394 to VXI Module

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.



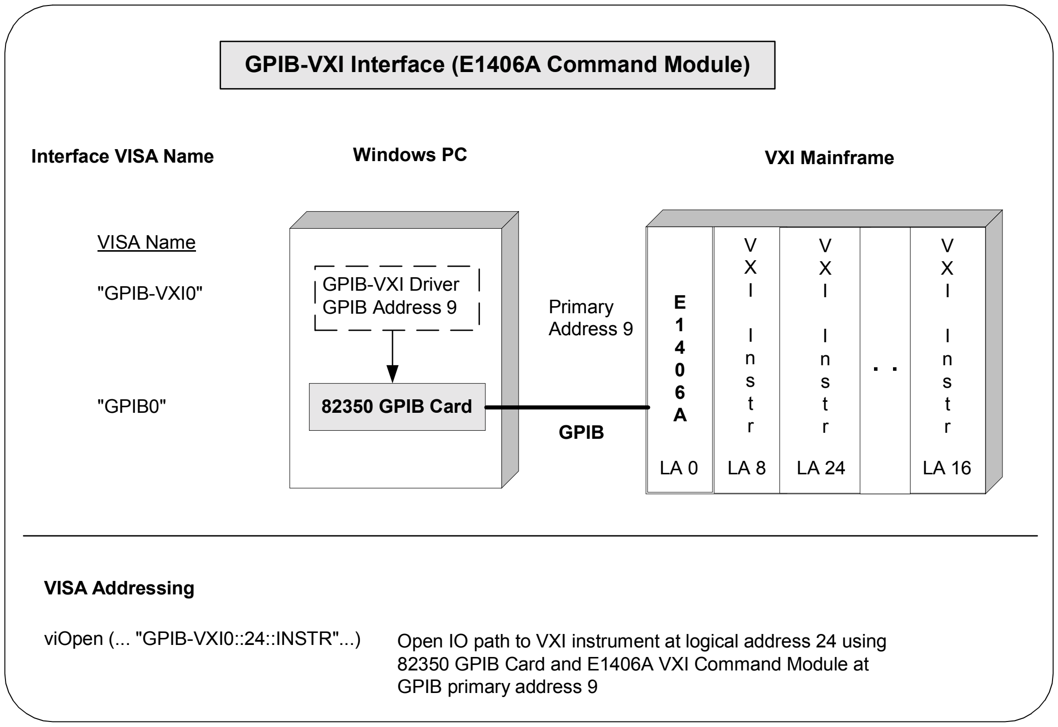
Example: GPIB-VXI (E1406A) Interface

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)
Interface VISA Name
Windows PC
VXI Mainframe
VISA Name
"GPIB-VXI0"
"GPIB0"
GPIB-VXI Driver GPIB Address 9
82350 GPIB Card
Primary Address 9
GPIB
E1406A
VXI Instr
LA 0
LA 8
LA 24
LA 16
VISA Addressing
viOpen (... "GPIB-VXI0::24::INSTR"...)
Open IO path to VXI instrument at logical address 24 using 82350 GPIB Card and E1406A VXI Command Module at GPIB primary address 9

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.

High-Level Memory Functions

This table summarizes the high-level memory functions.

Function	Description
`viIn8(`*vi, space, offset, val8***`);`**	Reads 8 bits of data from the specified offset.
`viIn16(`*vi, space, offset, val16***`);`**	Reads 16 bits of data from the specified offset.
`viIn32(`*vi, space, offset, val32***`);`**	Reads 32 bits of data from the specified offset.
`viOut8(`*vi, space, offset, val8***`);`**	Writes 8 bits of data to the specified offset.
`viOut16(`*vi, space, offset, val16***`);`**	Writes 16 bits of data to the specified offset.
`viOut32(`*vi, space, offset, val32***`);`**	Writes 32 bits of data to the specified offset.
`viMoveIn8(`*vi, space, offset, length, buf8***`);`**	Moves an 8-bit block of data from the specified offset to local memory.

Function	Description
viMoveIn16(*vi, space, offset, length, buf16***);**	Moves a 16-bit block of data from the specified offset to local memory.
viMoveIn32(*vi, space, offset, length, buf32***);**	Moves a 32-bit block of data from the specified offset to local memory.
viMoveOut8(*vi, space, offset, length, buf8***);**	Moves an 8-bit block of data from local memory to the specified offset.
viMoveOut16(*vi, space, offset, length, buf16***);**	Moves a 16-bit block of data from local memory to the specified offset.
viMoveOut32(*vi, space, offset, length, buf32***);**	Moves a 32-bit block of data from local memory to the specified offset.

Using **viIn** and **viOut**

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:

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**.

```
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:

`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.

```
  /* 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 () {
     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);
}
```

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.

```
   /*gpibvxih.c
   This example program uses the high-level memory
functions
   to read the id and device type registers of the device
at
   GPIB-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;
```

```
    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`**.

Low-Level Memory Functions

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

Function	Description
`viMapAddress(`*vi, mapSpace, mapBase, mapSize, access, suggested, address***`);`**	Maps the specified memory space.
`viPeek8(`*vi, addr, val8***`);`**	Reads 8 bits of data from address specified.
`viPeek16(`*vi, addr, val16***`);`**	Reads 16 bits of data from address specified.

Function	Description
`viPeek32(`*vi, addr, val32***`);`**	Reads 32 bits of data from address specified.
`viPoke8(`*vi, addr, val8***`);`**	Writes 8 bits of data to address specified.
`viPoke16(`*vi, addr, val16***`);`**	Writes 16 bits of data to address specified.
`viPoke32(`*vi, addr, val32***`);`**	Writes 32 bits of data to address specified.
`viUnmapAddress(`*vi***`);`**	Unmaps memory space previously mapped.

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:

`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:

```
/* 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);
```

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.

```
  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 **`viOpen`**.

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.

```
/*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);
}
```

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

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/voMoveOut`** 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];

  /*Open default resource manager and session to instr*/
  viOpenDefaultRM (&defaultRM);
  viOpen (defaultRM, VXI_INST, VI_NULL,VI_NULL, &vi);

/*    ====================================================
    Low level memory I/O = viPeek16 = direct memory
    dereference (when allowed)
    ====================================================*/

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

  /* ====================================================
    Using viPeek
    ====================================================*/

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

  /* Read device type 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\n",
     memArray[0]);
   printf (" viMoveIn16: Device Type Register = 0x%4X\n",
   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\n",
           memArray[0]);
   printf (" viMoveIn16: 2 ID Register = 0x%4X\n",
          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 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.

```
/* 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)\n '%s'\n", 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: ox%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.

Attribute	Description
`VI_ATTR_INTF_TYPE`	Interface type of the given session.
`VI_ATTR_INTF_NUM`	Board number for the given interface.
`VI_ATTR_TMO_VALUE`	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.
`VI_ATTR_INTF_INST_NAME`	Human-readable text describing the given interface.
`VI_ATTR_DMA_ALLOW_EN`	Specifies whether I/O accesses should use DMA (**`VI_TRUE`**) or Programmed I/O (**`VI_FALSE`**).

VXI and GPIB-VXI Specific MEMACC Attributes

The following attributes, most of which are read/write, provide memory window control information.

Attribute	Description
`VI_ATTR_VXI_LA`	Logical address of the local controller.
`VI_ATTR_SRC_INCREMENT`	Used in **`viMoveIn`*`xx`*** operation to specify how much the source offset is to be incremented after every transfer. The default value is 1 and the **`viMoveIn`*`xx`*** operation moves from consecutive elements. If this attribute is set to 0, the **`viMoveIn`*`xx`*** operation will always read from the same element, essentially treating the source as a FIFO register.

Attribute	Description
`VI_ATTR_DEST_INCREMENT`	Used in **viMoveOut*xx*** operation to specify how much the destination offset is to be incremented after every transfer. The default value is 1 and the **viMoveOut*xx*** operation moves into consecutive elements. If this attribute is set to 0, the **viMoveOut*xx*** operation will always write to the same element, essentially treating the destination as a FIFO register.
`VI_ATTR_WIN_ACCESS`	Specifies modes in which the current window may be addressed: not currently mapped, through the **viPeek*xx*** or **viPoke*xx*** operations only, or through operations and/or by directly de-referencing the address parameter as a pointer.
`VI_ATTR_WIN_BASE_ADDR`	Base address of the interface bus to which this window is mapped.
`VI_ATTR_WIN_SIZE`	Size of the region mapped to this window.
`VI_ATTR_SRC_BYTE_ORDER`	Specifies the byte order used in high-level access operations, such as **viIn*xx*** and **viMoveIn*xx*,** when reading from the source.
`VI_ATTR_DEST_BYTE_ORDER`	Specifies the byte order used in high level access operations, such as **viOut*xx*** and **viMoveOut*xx***, when writing to the destination.
`VI_ATTR_WIN_BYTE_ORDER`	Specifies the byte order used in low-level access operations, such as **viMapAddress**, **viPeek*xx***, and **viPoke*xx***, when accessing the mapped window.
`VI_ATTR_SRC_ACCESS_PRIV`	Specifies the address modifier used in high-level access operations, such as **viIn*xx*** and **viMoveIn*xx***, when reading from the source.
`VI_ATTR_DEST_ACCESS_PRIV`	Specifies address modifier used in high-level access operations such as **viOut*xx*** and **viMoveOut*xx***, when writing to destination.
`VI_ATTR_WIN_ACCESS_PRIV`	Specifies address modifier used in low-level access operations, such as **viMapAddress, viPeek*xx***, and **viPoke*xx***, when accessing the mapped window.

GPIB-VXI Specific MEMACC Attributes

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

Attribute	Description
`VI_ATTR_INTF_PARENT_NUM`	Board number of the GPIB board to which the GPIB-VXI is attached.
`VI_ATTR_GPIB_PRIMARY_ADDR`	Primary address of the GPIB-VXI controller used by the session.
`VI_ATTR_GPIB_SECONDARY_ADDR`	Secondary address of the GPIB-VXI controller used by the session.

MEMACC Resource Event Attribute

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

Attribute	Description
`VI_ATTR_EVENT_TYPE`	Unique logical identifier of the event.
`VI_ATTR_STATUS`	Return code of the asynchronous I/O operation that has completed.
`VI_ATTR_JOB_ID`	Job ID of the asynchronous I/O operation that has completed.
`VI_ATTR_BUFFER`	Address of a buffer used in an asynchronous operation.
`VI_ATTR_RET_COUNT`	Actual number of elements that were asynchronously transferred.

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.

`VI_ATTR_WIN_ACCESS Settings`

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

Setting	Description
`VI_NMAPPED`	Specifies that the window is not mapped.
`VI_USE_OPERS`	Specifies that the window is mapped and you can only use the low-level memory functions to access the data.
`VI_DEREF_ADDR`	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.

G.2 SICL Library

Standard Instrument Control Library

User’s Guide

Notice

Warranty Information

U.S. Government Restricted Rights

Printing History

This is the seventh edition of the *Standard Insrtument 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 5 - July 1998

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

Error Handler Example 62
Using Locking 64
Lock Actions 65
Locking in a Multi-user Environment 66
Locking Example 67

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

Generic I/O Error....206
Bad Parameter....207
Where to Find Additional Information....208

10. SICL Language Reference

IABORT....212
IBLOCKCOPY....213
IBLOCKMOVEX....215
ICAUSEERR....217
ICLEAR....218
ICLOSE....219
IDEREFPTR....220
IFLUSH....221
IFREAD....223
IFWRITE....225
IGETADDR....226
IGETDATA....227
IGETDEVADDR....228
IGETERRNO....229
IGETERRSTR....231
IGETGATEWAYTYPE....232
IGETINTFSESS....233
IGETINTFTYPE....234
IGETLOCKWAIT....235
IGETLU....236
IGETLUINFO....237
IGETLULIST....238
IGETONERROR....239
IGETONINTR....240
IGETONSRQ....241
IGETSESSTYPE....242
IGETTERMCHR....243
IGETTIMEOUT....244
IGPIBATNCTL....245
IGPIBBUSADDR....246
IGPIBBUSSTATUS....247
IGPIBGETT1DELAY....249

IGPIBLLO 250
IGPIBPASSCTL 251
IGPIBPPOLL 252
IGPIBPPOLLCONFIG 253
IGPIBPPOLLRESP 254
IGPIBRENCTL 255
IGPIBSENDCMD 256
IGPIBSETT1DELAY 257
IGPIOCTRL 258
IGPIOGETWIDTH 263
IGPIOSETWIDTH 264
IGPIOSTAT 266
IHINT 269
IINTROFF 271
IINTRON 272
ILANGETTIMEOUT 273
ILANTIMEOUT 274
ILOCAL 277
ILOCK 278
IMAP 280
IMAPX 283
IMAPINFO 286
IONERROR 288
IONINTR 291
IONSRQ 293
IOPEN 294
IPEEK 296
IPEEKX8, IPEEKX16, IPEEKX32 297
IPOKE 298
IPOKEX8, IPOKEX16, IPOKEX32 299
IPOPFIFO 300
IPRINTF 302
IPROMPTF 312
IPUSHFIFO 313
IREAD 315
IREADSTB 317
IREMOTE 318
ISCANF 319

ISERIALBREAK 329
ISERIALCTRL 330
ISERIALMCLCTRL 334
ISERIALMCLSTAT 335
ISERIALSTAT 336
ISETBUF 340
ISETDATA 342
ISETINTR 343
ISETLOCKWAIT 351
ISETSTB 352
ISETUBUF 353
ISWAP 355
ITERMCHR 357
ITIMEOUT 358
ITRIGGER 359
IUNLOCK 361
IUNMAP 362
IUNMAPX 364
IVERSION 365
IVXIBUSSTATUS 366
IVXIGETTRIGROUTE 369
IVXIRMINFO 370
IVXISERVANTS 372
IVXITRIGOFF 373
IVXITRIGON 375
IVXITRIGROUTE 377
IVXIWAITNORMOP 379
IVXIWS 380
IWAITHDLR 381
IWRITE 383
IXTRIG 384
_SICLCLEANUP 387

A. The SICL Files

B. Updating HP-UX 9 SICL Applications

Building SICL Applications on HP-UX 11i 399

Linking with the Archive
    Library on HP-UX 9 400

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

1

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 languag 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*
  - *Managing Systems and Workgroups: A Guide for HP-UX System Administrators*

- 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 (HPIB)*

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

- 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*

2

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.

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.

```
/* 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.

itimeout 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.c library.sl -lsicl
```

on Linux, use

```
gcc -c library.c -lsicl
ld -shared -o library.so library.o
gcc -o y y.c -L. -llibrary -lsicl
```

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 (`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 `man` command followed by the SICL function call or utility:

```
man  name
```

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

```
man iprintf
man ipromptf
man iread
```

Examples of SICL utilities:

```
man ipeek
man iread
man ivxisc
```

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."

3

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:

```
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 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 that corresponds to a specific interface. The `symbolic name` is a string which uniquely describes the interface.

The following are valid interface addresses:

`7`	Interface card at logical unit 7.
`hpib`	GPIB interface with the symbolic name hpib.
`hpib2`	Second GPIB interface with the symbolic name hpib2.

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).

Addressing Commander Sessions

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:

`hpib,cmdr`	GPIB commander session.
`7,cmdr`	Commander session on interface at logical unit 7.

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 `iprintf` function formats according to the *format* string and sends data to the session specified by *id*:

  `iprintf (`*id, format [,arg1][,arg2][,...]*`);`

- The `iscanf` function receives data from the session specified by *id* and converts the data according to the *format* string:

  `iscanf(`*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:

  `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:

`%[`*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`

Flag	Description
@1	Converts to a IEEE 488.2 NR1 number.
@2	Converts to a IEEE 488.2 NR2 number.
@3	Converts to a IEEE 488.2 NR3 number.
@H	Converts to a IEEE 488.2 hexadecimal number.
@Q	Converts to a IEEE 488.2 octal number.
@B	Converts to a IEEE 488.2 binary number.
+	Prefixes number with sign (+ or -).
–	Left justifies result.
space	Prefixes number with blank space if positive or with - if negative.
#	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.
0	Causes the left pad character to be a zero for all numeric conversion types.

The following example converts `numb` into a IEEE 488.2 floating point number (NR2) and sends it to the session specified by `id`:

```
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`:

```
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`:

```
float numb = 26.9345;
iprintf (id, "%.2f", numb);
```

Sends : `26.93`

,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 `,dd` where `dd` 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`:

```
int list[5]={101,102,103,104,105};
iprintf (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:

Argument Modifiers

Argument Modifier	Conversion Character	Description
h	d, i	Corresponding argument is a short integer.
h	f	Corresponding argument is a float for iprintf or a pointer to a float for iscanf.
l	d, i	Corresponding argument is a long integer.
l	b, B	Corresponding argument is a pointer to an array of long integers.
l	f	Corresponding argument is a double for iprintf or a pointer to a double for iscanf.
w	b, B	Corresponding argument is a pointer to an array of short integers.
z	b, B	Corresponding argument is pointer to an array of floats.
Z	b, B	Corresponding argument is a pointer to an array of doubles.

Conversion Characters. The conversion characters for sending and receiving formatted I/O are different. The following tables summarize the conversion characters for each:

`iprintf` and `ipromptf` Conversion Characters

Conversion Character	Description
d, i	Corresponding argument is an integer
f	Corresponding argument is a double.
b, B	Corresponding argument is a pointer to an arbitrary block of data.
c, C	Corresponding argument is a character.
t	Controls whether the END indicator is sent with each LF character in the format string.
s, S	Corresponding argument is a pointer to a null terminated string.
%	Sends an ASCII percent (%) character.
o, u, x, X	Corresponding argument is an unsigned integer.
e, E, g, G	Corresponding argument is a double.
n	Corresponding argument is a pointer to an integer.
f	Corresponding argument is a pointer to a FILE descriptor opened for reading.

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:

```
long int size = 1024;
char data [1024];
   .
   .
iprintf (id, "%*b", size, data);
```

Sends 1024 characters of block data.

`iscanf` and `ipromptf` Conversion Characters

Conversion Character	Description
d, i, n	Corresponding argument must be a pointer to an integer.
e, f, g	Corresponding argument must be a pointer to a float.
c	Corresponding argument is a pointer to a character sequence.
s, S, t	Corresponding argument is a pointer to a string.
o, u, x	Corresponding argument must be a pointer to an unsigned integer.
[	Corresponding argument must be a character pointer.
F	Corresponding argument is a pointer to a FILE descriptor opened for writing.

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:

```
char  data[180];
iscanf (id, "%s", data);
```

Formatted I/O Example

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.

```
/* formatio.c
   This example program makes a multimeter measurement
   with a comma separated list passed with formatted I/O
   and prints the results */
#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);
}
```

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:

`ifread`	Obtains raw data directly from the read formatted I/O buffer. This is the same buffer that iscanf uses.
`ifwrite`	Writes raw data directly to the write formatted I/O buffer. This is the same buffer that iprintf uses.
`iprintf`	Converts data via a format string and writes the arguments appropriately.
`iscanf`	Reads data, converts this data via a format string, and assigns the values to your arguments.
`ipromptf`	Sends, then receives, data from a device/instrument. It also converts data via format strings that are identical to iprintf and iscanf. The advantage of this function is that the iprintf and iscanf parts are done together.
`iflush`	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.
`isetbuf`	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.
`isetubuf`	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.

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.

```
/* 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`.

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.

For example:

```
...
iintroff ()
ionintr (vxi, trigger_handler);
isetintr (vxi, I_INTR_TRIG, I_TRIG_TTL0 | I_TRIG_TTL7);
...
ivxitrigon (vxi, I_TRIG_TTL0);
while (!done)
  iwaithdlr (0);
iintron ();
...
```

Asynchronous Events and Unix Signals

Note 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:

```
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.

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.

```
.
.
/* 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.

```
/* 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);
   /* 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);
}

child ()
{
   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:

```
   int ionerror (proc);
   void (*proc)();

Where:

   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.

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);
  iprintf (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:

```
  /* 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\n", igeterrstr (error));
  exit (1);
}

main () {
  INST dvm;
  double res;

  ionerror (err_handler);
  dvm = iopen ("hpib,16");
  itimeout (dvm, 10000);
  iprintf (dvm, "%s\n", "MEAS:VOLT:DC?");
  iscanf (dvm, "%lf", &res);
  printf ("Result is %f\n", 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 `ilock` function is used to **lock** an interface or device. The SICL `iunlock` 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.

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);
}
```

4

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 an 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:

```
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.

`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.
`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.
`ireadstb`	Performs a GPIB serial poll (SPOLL).
`itrigger`	Performs an addressed GPIB group execute trigger (GET).
`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 `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 **status byte** by using the `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.

The following example opens a interface session with the GPIB interface:

```
INST hpib;
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\n", "REM",
      "SRQ", "NDC", "SYS", "ACT", "TLK", "LTN", "ADDR");
   printf("%2d%5d%5d%5d%5d%5d%5d%6d\n", rem, srq, ndac,
       sysctlr, actctlr, talker, listener, addr);
  return 0;
}
```

Communicating with Devices via Interface Sessions

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");
  /* 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];
  sdiv  = pre[2] * pre[4] / 10;
  delay = (pre[2] / 2 - pre[6]) * pre[4] + pre[5];

  /* retrieve the scope’s ID string */
  ipromptf(scope, "*IDN?\n", "%s", id_str);

   /*  print the statistics about the data */
  printf("\nOscilloscope 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");
  /* 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:

`hpib,cmdr`	A commander session with the `hpib` symbolic name.
`hpib2,cmdr`	A commander session with the `hpib2` 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.

The following example opens a commander session the GPIB interface:

```
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.

GPIB Commander Session Interrupts

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 can not be used on other interfaces and, therefore, becomes less portable.

SICL GPIB Functions

Function Name	Action
igpibatnctl	Sets or clears the ATN line
igpibbusaddr|	Change bus address
igpibbusstatus	Return requested bus data
igpibgett1delay	Retrieves the T1 delay setting on the GPIB interface
	Sets bus in Local Lockout Mode
igpibllo	Passes active control to specified address
igpibpassctl	Performs a parallel poll on the bus
igpibppoll	Configures device for PPOLL response
igpibppollconfig	Sets PPOLL state
igpibppollresp	Sets or clears the REN line
igpibrenctl	Sends data with ATN line set
igpibsendcmd	Sets the T1 delay on the GPIB interface
igpibsett1delay	

Appendix H

Device Manuals

H.1 Function Generator

H.1.1 Quick Reference

Agilent 33120A
Function / Arbitrary Waveform Generator
Quick Reference Guide

Front-Panel Menu Reference

Use ***Recall Menu*** as a shortcut to recall the last command executed.

A: MODulation MENU

1: AM SHAPE ⇒ 2: AM SOURCE ⇒ • • • ⇒ 9: FSK RATE ⇒ 10: FSK SRC

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: BURST CNT	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 a burst (-360 to +360 degrees).
7: BURST SRC	Selects an internal or external gate source for burst modulation.
8: FSK FREQ	Sets the FSK "hop" frequency.
9: FSK RATE	Selects 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 ⇒ [2: POINTS] ⇒ • • • ⇒ [6: SAVE AS] ⇒ 7: DELETE

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 ⇒ • • • ⇒ 5: COMMA ⇒ 6: 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	Retrieves 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	Selects even, odd, or no parity for RS-232 operation.
5: LANGUAGE	Verifies the interface language: SCPI.

F: CALibration MENU *

1: SECURED ⇒ [1: UNSECURED] ⇒ [2: CALIBRATE] ⇒ 3: CAL COUNT ⇒ 4: MESSAGE

1: SECURED	The function generator is secured against calibration; enter code to unsecure.
[1: UNSECURED]	The function generator is unsecured for calibration; enter code to secure.
[2: CALIBRATE]	Performs individual calibrations; must be UNSECURED.
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.

Agilent Technologies

- Square brackets (**[ ]**) indicate optional keywords or parameters.
- Braces (**{ }**) enclose parameters within a command string. Default parameters are shown in **`bold`**.
- Triangle brackets (**< >**) indicate that you must substitute a value for the enclosed parameter.

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?

*SAV {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?

*PSC {0|1}
*PSC?

*SRE <enable value>
*SRE?

*STB?

*WAI
```

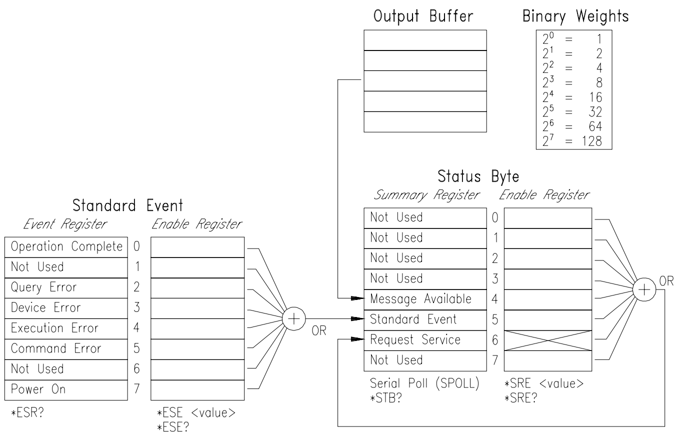
Calibration Commands

(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)



33120-90009

IEEE-488.2 Common Commands

(see page 209 in User's Guide)

```
*CLS                          *RST

*ESE <enable value>           *SAV {0|1|2|3}
*ESE?                         *RCL {0|1|2|3}

*ESR?                         *SRE <enable value>
                              *SRE?
*IDN?
                              *STB?
*OPC
                              *TRG
*OPC?
                              *TST?
*PSC {0|1}
*PSC?                         *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.

-330, "Self-test failed" The `*TST?` command failed.

-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.

Power-On and Reset State

The parameters marked with a bullet (•) are stored in ***non-volatile*** *memory. The factory settings are shown.*

Output Configuration	**Power-On/Reset State**
Function	Sine wave
Frequency	1 kHz
Amplitude (into 50 ohms)	100 mV peak-to-peak
Offset	0.00 Vdc
Output Units	Volts peak-to-peak
Output Termination	50 ohms
Modulation	**Power-On/Reset State**
AM Carrier Waveform	1 kHz Sine wave
AM Modulating Waveform	100 Hz Sine wave
AM Depth	100%
FM Carrier Waveform	1 kHz Sine wave
FM Modulating Waveform	10 Hz Sine wave
FM Peak Frequency Deviation	100 Hz
Burst Carrier Frequency	1 kHz Sine wave
Burst Count	1 cycle
Burst Rate	100 Hz
Burst Starting Phase	0 degrees
FSK Carrier Waveform	1 kHz Sine wave
FSK “Hop” Frequency	100 Hz Sine wave
FSK Rate	10 Hz
Modulation State	Off
Sweep Start / Stop Frequency	100 Hz / 1 kHz
Sweep Time	1 second
Sweep Mode	Linear
System-Related Operations	**Power-On/Reset State**
• Power-Down Recall	• Disabled
Display Mode	On
• Comma Separators	• On
Triggering Operations	**Power-On/Reset State**
Trigger Source	Internal
Input/Output Configuration	**Power-On/Reset State**
• GPIB Address	• 10
• Interface	• GPIB (IEEE-488)
• Baud Rate	• 9600 baud
• Parity	• None (8 data bits)
Calibration	**Power-On/Reset State**
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

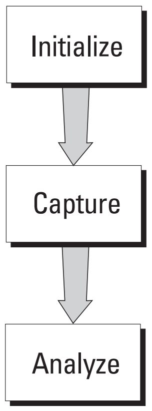
For Safety information, Warranties, and Regulatory information, see the pages behind the Index.

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.



Initialize

To ensure consistent, repeatable performance, you need to start the program, controller, and oscilloscope in a known state. Without correct initialization, your program may run correctly in one instance and not in another. This might be due to changes made in configuration by previous program runs or from the front panel of the oscilloscope.

- 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.

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).

1 **Introduction to Programming**

2 **Programming Getting Started**

3 **Programming over GPIB**

4 **Programming over RS-232-C**

5 **Programming and Documentation Conventions**

6 **Status Reporting**

7 **Installing and Using the Programmer's Reference**

8 **Programmer's Quick Reference**

Index

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 DIGitize 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

1

Introduction to Programming

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:

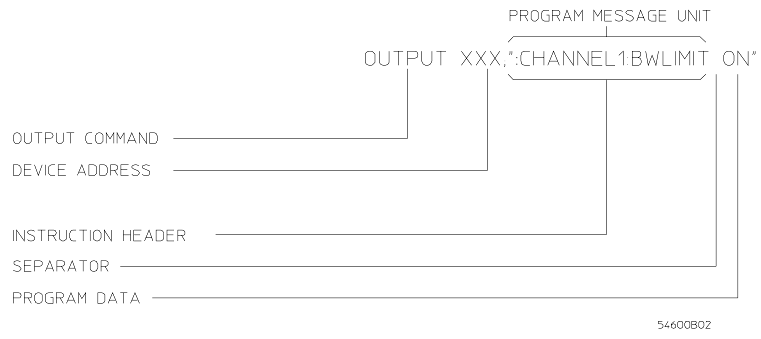
```
OUTPUT < device address > ;":CHANNEL1:BWLIMIT ON"<terminator>
```

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.

Figure 1-1



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:

```
<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:

```
<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:

```
:<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:

```
*<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>:<function><separator><data>;
     <function><separator><data><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.”

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).

New Line Terminator Functions

The NL (New Line) terminator has the same function as an EOS (End Of String) and EOT (End Of Text) terminator.

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
```

Combining Compound and Simple Commands

Multiple commands may be any combination of compound and simple commands.

2

Programming Getting Started

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.

Languages for Programming Examples

The programming examples in this guide are written in Agilent BASIC, C, or SICL C.

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 (100µs/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
```

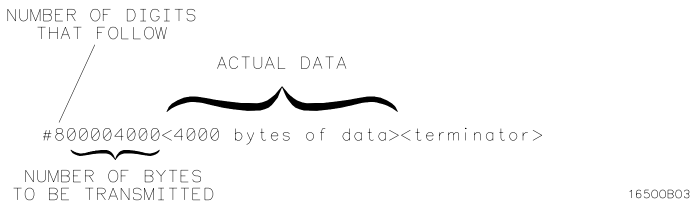
After running this program, the controller displays:

```
.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:


NUMBER OF DIGITS THAT FOLLOW
ACTUAL DATA
#800004000<4000 bytes of data><terminator>
NUMBER OF BYTES TO BE TRANSMITTED
16500B03

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.

3

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

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.

Restore Front-Panel Control

Cycling power also restores front panel control.

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.

8

Programmer’s Quick Reference

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>.
I "or"	Indicates a choice of one element from a list. For example, <A> I <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.

Command	Query	Options and Query Returns
:ACQuire:COMPlete <complete_argument>	:ACQuire:COMPlete?	<complete_argument> ::= 0 to 100; an integer in NR1 format
:ACQuire:COUNt <count_argument>	:ACQuire:COUNT?	<count_argument> ::= 8, 64, or 256; an integer in NR1 format
n/a	**:ACQuire:POINts?**	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.
n/a	**:ACQuire:SETup?**	ACQuire:TYPE{NORM | AVER | PEAK}; COUNt<count_argument>; (8, 64, or 256; an integer in NR1 format); POINts<points_argument>; 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. COMPlete<complete_argument> 0 to 100; an integer in NR1 format
:ACQuire:TYPE <acq_type>	:ACQuire:TYPE?	<acq_type> ::= {NORMal | AVERage | PEAK}
:AStore	n/a	n/a
:AUToscale	n/a	n/a
:BLANk <display>	n/a	<display> ::= {CHAN <n> | PMEM{1 | 2}} for the 54600/01/02/03/15/16 {CHAN <n> | PMEM{1 | 2} | EXTernal} 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
:CHANnel<n>:BWLimit {ON | OFF}	:CHANnel<n>:BWLimit?	{ON | OFF} <n> ::= 1 or 2; an integer in NR1 format
:CHANnel<n>:COUPling {AC | DC | GND}	:CHANnel<n>:COUPling?	{AC | DC | GND} <n> ::= 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
:CHANnel<n>:INPut {FIFTy | ONEMeg}	:CHANnel<n>:INPut?	{FIFTy | ONEMeg} <n> ::= 1 or 2; an integer in NR1 format
:CHANnel<n>:INVert {ON | OFF}	:CHANnel<n>:INVert?	{ON | OFF} <n> ::= 1 or 2; an integer in NR1 format
:CHANnel:MATH {OFF | PLUS | SUBTract}	:CHANnel:MATH?	{OFF | PLUS | SUBTract}
:CHANnel<n>:OFFSet <offset_argument>	:CHANnel<n>:OFFSet?	<offset_argument> ::= offset value in volts in <NR3> format. <n> ::= 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
:CHANnel<n>:PMODe {AUTo | MANual}	:CHANnel<n>:PMODe?	{AUT | MAN} <n> ::= 1 or 2; an integer in NR1 format

Command	Query	Options and Query Returns
:CHANnel<n>:PROBe <attenuation>	:CHANnel<n>:PROBe?	<attenuation> ::= X1, X10, X100 for 51600/01/02/03 X1, X10, X20, X100 for the 54610/15/16 <n> ::= 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
:CHANnel<n>:PROTect {OFF | ON}	:CHANnel<n>:PROTect?	{OFF | ON} <n> ::= 1 or 2; an integer in NR1 format
:CHANnel<n>:RANGe <range_argument>	:CHANnel<n>:RANGe?	<range_argument> ::= Full-scale range value for channels 1 or 2 in NR3 format, and {LOW | HIGH} for channels 3 or 4.

n/a	**:CHANnel<n>:SETup?**	For 54600/01/02/03 channels 1 and 2: CHANnel<n>:RANGe <range>; OFFSet <offset>; COUPling {AC | DC | GND}; BWLimit {ON | OFF}; INVert {ON | OFF}; VERNier {ON | OFF}; PROBe {X1 | X10 | X100}
		
		For 54610/15/16 channel 1: CHANnel1:RANGe <range>; OFFSet <offset>; COUPling {AC | DC | GND}; BWLimit {ON | OFF}; INVert {ON | OFF}; VERNier {ON | OFF}; PROBe {X1 | X10 | X20 | X100}; PMODe {AUT | MAN}; INPut {FIFTy | ONEMeg}; PROTect {OFF | ON}
		
		For 54610/15/16 channel 2: CHANnel2:RANGe <range>; OFFSet <offset>; COUPling {AC | DC | GND}; BWLimit {ON | OFF}; INVert {ON | OFF}; VERNier {ON | OFF}; PROBe {X1 | X10 | X20 | X100}; PMODe {AUT | MAN}; INPut {FIFTy | ONEMeg}; PROTect {OFF | ON}; SKEW <skew_value>
		
		For 54601/02 channels 3 or 4: CHANnel<n>:RANGe {HIGH | LOW}; OFFSet <offset>; COUPling {DC | GND}; PROBe {X1 | X10 | X100}

:CHANnel2:SKEW <skew_argument>	:CHANnel2:SKEW?	<skew_argument> ::= the skew value in seconds in <NR3> format
:CHANnel<n>:VERNier {ON | OFF}	:CHANnel<n>:VERNier?	{ON | OFF}
***CLS**	n/a	n/a
:DIGitize CHANnel<n>, [CHANnel<n>]	n/a	n/a
:DITHer {ON | OFF}	:DITHer?	{ON | OFF}
:DISPLAY:COLumn <number>	:DISPLAY:COLumn?	<number> ::= 0 through 63; an integer in NR1 format
:DISPlay:CONNect {ON | OFF}	:DISPlay:CONNect?	{ON | OFF}
:DISPlay:DATA **<binary block_data>**	:DISPlay:DATA?	<binary block_data> ::= 16256 bytes of data in IEEE 488.2 # format

Command	Query	Options and Query Returns
:DISPlay:GRID {ON | OFF | SIMPle | TV}	:DISPlay:GRID?	{ON | OFF | SIMPle | TV}
:DISPlay:INVerse {ON | OFF}	:DISPlay:INVerse?	{ON | OFF}
:DISPlay:LINE <string>	n/a	<string> ::= any series of ASCII characters enclosed in quotation marks
:DISPlay:PALette <palette_number>	:DISPlay:PALette?	<palette_number> ::= 0 through 6; an integer in NR1 format
: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> (comand) ::= 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

Command	Query	Options and Query Returns

n/a	**:DISPlay:SETup?**	:DISPlay:ROW <row_number>;
		    <row_number> ::= 1...20; an integer in NR1 format
		COLumn <column_number>;
		    <column_number> ::= 0...63; an integer in NR1 format
		INVerse <inverse>;
		    <inverse> ::= {ON | OFF}
		GRID <grid>;
		    <grid> ::= {ON | OFF}
		SOURce <source>;
		    <source> ::= {PMEMory1 | PMEMory2}
		CONNect <connect_status>
		    <connect_status> ::= {ON | OFF}
		PALette <palette_number> (54616C only)
		    <palette_number> ::= 0...6; an integer in NR1 format

:DISPlay:SOURce <value>	:DISPlay:SOURce?	<value> ::= {PMEMory1 | PMEMory2}
:DISPlay:TEXT BLANk	n/a	n/a
:ERASe <value>	n/a	<value> ::= {PMEMory1 | PMEMory2}
***ESE** <mask_argument>	*ESE?	<mask_argument> ::= 0...255; an integer in NR1 format Bit Weight Enables 7 128 NOT USED 6 64 URQ - User Request 5 32 CME - Command Error 4 16 EXE - Execution Error 3 8 DDE - Device Dependent Error 2 4 QYE - Query Error 1 2 TRG - Trigger Query 0 1 OPC - Operation Complete
n/a	***ESR?**	<status> ::= 0...255; an integer in NR1 format
:EXTernal:COUPling {DC | AC | GND}	:EXTernal:COUPling?	{DC | AC | GND}
:EXTernal:INPut {FIFTy | ONEMEG}	:EXTernal:INPut?	{FIFTy | ONEMEG}
:EXTernal:OFFSet <offset_argument>	:EXTernal:OFFSet?	<offset_argument> ::= offset value in volts in NR3 format
:EXTernal:PMODe {AUTo | MANual}	:EXTernal:PMODe?	{AUTo | MANual}
:EXTernal:PROBe <attenuation>	:EXTernal:PROBe?	<attenuation> ::= {X1 | X10 | X20 | X100} for the 54610/15/16
:EXTernal:PROTect {OFF | ON}	:EXTernal:PROTect?	{OFF | ON}

Command	**Query**	**Options and Query Returns**

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
n/a	**:FUNCtion2:PEAKs?** {FREQ1 | DB1 | FREQ2 | DB2}	<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	{RECTangular | HANNing | FLATtop | EXPonent}

Command	Query	Options and Query Returns
n/a	***IDN?**	HEWLETT-PACKARD,<model>, 0, X.X <model> ::= the model number of the instrument <X.X> ::= the software revision of the instrument
n/a	***LRN?**	<learn_string> ::= a maximum of 218 bytes of data in IEEE 488.2 # format
:MASK:CREATe	n/a	n/a
:MASK:DATA	:MASK:DATA?	<header> ::= block header that contains the ASCII characters #8000998 and is sent prior to the data. <mask_data> ::= 998 bytes of data that represent the currently selected mask template.
:MASK:DESTination {TRACe | PRINter}	:MASK:DESTination?	{TRACe | PRINter}
:MASK:FAILmode {IN | OUT}	:MASK:FAILmode?	{IN | OUT}
:MASK:INCRement {ON | OFF}	:MASK:INCRement?	{ON | OFF}
:MASK:NUMBer <number>	:MASK:NUMBer?	<number> ::= memory (1 or 2)
:MASK:POSTfailure {RUN | STOP}	:MASK:POSTfailure?	{RUN | STOP}
:MASK:SAVE {ON | OFF}	:MASK:SAVE?	{ON | OFF}
n/a	**:MASK:STATistics?**	<compares, failures, failure %> ::= current number of mask tests performed number of failures detected percentage of failures
:MASK:TEST {ON | OFF}	:MASK:TEST?	{ON | OFF}
:MASK:TOLerance <value>	:MASK:TOLerance?	<value> ::= the tolerance used when creating a mask template. The entered value can be from 0.00 to 20.0 percent.
n/a	**:MEASure:ALL?**	<value list> ::= <FREQ result>, <PERIOD result>, <+ WID result>, <- WID result>, <RISE result>, <FALL result>, <VPP result>, <DUTY CYCLE result>, <VRMS result>, <VMAX result>, <VMIN result>, <VTOP result>, <VBASE result>, <VAVG result>, <VAMP result>, <Vovershoot result>, <Vpreshoot result> <result> ::= individual measurement results in NR3 format
:MEASure:DEFine DELay <edge1>,<edge2>	:MEASure:DEFine? DELay	<edgeN> ::= the edge selection for channels 1 and 2. N is the selected edge number (1 to 5).
:MEASure:DELay	:MEASure:DELay?	<return_value> ::= floating point number delay time in seconds in NR3 format
:MEASure:DUTYcycle	:MEASure: DUTYcycle?	<return_value> ::= ratio of positive pulse width to period in NR3 format

Command	Query	Options and Query Returns
:MEASure:FALLtime	:MEASure:FALLtime?	<return_value> ::= time in seconds between the 10% and 90% voltage levels in NR3 format
:MEASure:FREQuency	:MEASure:FREQuency?	<return_value> ::= frequency in Hertz in NR3 format
:MEASure:LOWer <voltage>	:MEASure:LOWer?	<voltage> ::= the user-defined lower threshold in volts in NR3 format
:MEASure:NWIDth	:MEASure:NWIDth?	<return_value> ::= negative pulse width in seconds in NR3 format
:MEASure:OVERshoot	:MEASure:OVERshoot?	<voltage> ::= the percent of the overshoot of the selected waveform in NR3 format
:MEASure:PERiod	:MEASure:PERiod?	<return_value> ::= waveform period in seconds in NR3 format
:MEASure:PHASe	:MEASure:PHASe?	<return_value> ::= the phase angle value in degrees in NR3 format
:MEASure:PREShoot	:MEASure:PREShoot?	<return_value> ::= the percent of preshoot of the selected waveform in NR3 format
:MEASure:PSTArt <value>	:MEASure:PSTArt?	<value> ::= the relative position of time marker 1 in degrees in NR3 format
:MEASure:PSTOp <value>	:MEASure:PSTOp?	<value> ::= the relative position of time marker 2 in degrees in NR3 format
:MEASure:PWIDth	:MEASure:PWIDth?	<return_value> ::= width of positive pulse in seconds in NR3 format
:MEASure:RISEtime	:MEASure: RISEtime?	<return_value> ::= rise time in seconds in NR3 format
:MEASure:SCRatch	n/a	n/a
:MEASure:SET100	n/a	n/a
:MEASure:SET360	n/a	n/a
:MEASure:SHOW {ON | OFF}	:MEASure:SHOW?	{ON | OFF}
:MEASure:SOURce CHANnel <n>	:MEASure:SOURce?	<n> ::= 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
n/a	**:MEASure:TDELta?**	<return_value> ::= time difference in seconds between start and stop markers in NR3 format
:MEASure:THResholds {T1090 | T2080 | VOLTage}	:MEASure:THResholds?	{T1090 | T2080 | VOLTage}
:MEASure:TSTArt <value>	:MEASure:TSTArt?	<value> ::= time at the start marker in seconds in NR3 format
:MEASure:TSTOp <value>	:MEASure:TSTOp?	<value> ::= time at the stop marker in seconds in NR3 format
n/a	**:MEASure:TVOLt** <tvolt_argument>, <slope><occurrence>	<tvolt_argument> ::= positive or negative voltage level that the waveform must cross. <slope> ::= direction of the waveform when <tvolt_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**
:MEASure:UPPer <voltage>	:MEASure:UPPer?	<voltage> ::= the user-defined upper threshold in volts in NR3 format
:MEASure:VAMPlitude	:MEASure:VAMPlitude?	<return_value> ::= the amplitude of the selected waveform in volts in NR3 format
:MEASure:VAVerage	:MEASure:VAVerage?	<return_value> ::= calculated average voltage in NR3 format
:MEASure:VBASe	:MEASure:VBASe?	<base_voltage> ::= voltage at the base of the selected waveform in NR3 format
n/a	**:MEASure:VDELta?**	<return_value> ::= delta V value in volts in NR3 format
:MEASure:VMAX	:MEASure:VMAX?	<return_value> ::= maximum voltage of the selected waveform in NR3 format
:MEASure:VMIN	:MEASure:VMIN?	<return_value> ::= minimum voltage of the selected waveform in NR3 format
:MEASure:VPP	:MEASure:VPP?	<return_value> ::= voltage peak to peak in NR3 format
:MEASure:VPSTArt <value>	:MEASure:VPSTArt?	<value> ::= the relative position of voltage marker 1 in percent in NR3 format
:MEASure:VPSTOp <value>	:MEASure:VPSTOp?	<value> ::= the relative position of voltage marker 2 in percent in NR3 format
:MEASure:VRMS	:MEASure:VRMS?	<return_value> ::= calculated dc RMS voltage in NR3 format
:MEASURE:VSTArt <vstart_argument>	:MEASure:VSTArt?	<vstart_argument> ::= voltage value for VMarker 1 in NR3 format <return_value> ::= voltage at VMarker 1 in NR3 format
:MEASure:VSTOp <vstop_argument>	:MEASure:VSTOp?	<vstop_argument> ::= voltage value for VMarker 2 in NR3 format <return_value> ::= voltage at VMarker 2 in NR3 format
n/a	**:MEASure:VTIMe** <vtime_argument>	<vtime_argument> ::= displayed time from trigger in seconds in NR3 format <return_value> ::= voltage at the specified time in NR3 format
:MEASure:VTOP	:MEASure:VTOP?	<return_value> ::= voltage at the top of the waveform in NR3 format

Command	Query	Options and Query Returns
:MENU <integer>	:MENU?	<integer> ::= 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 5 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
:MERGe <pixel memory>	n/a	<pixel memory> ::= {PMEMory1 | PMEMory2}
***OPC**	*OPC?	ASCII "1" is placed in the output queue when all pending device operations have completed.
n/a	***OPT?**	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
n/a	**:PRINt?** [enhancement]	[enhancement] ::= [HIRes [,PCLColor]] HIRes ::= contains both half-bright and full-bright display information PCLColor ::= color DeskJet selection only on 54616C
***RCL** <value>	n/a	<value> ::= {1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 }
***RST**	n/a	See reset values in the online Programmer's Reference.
:RUN	n/a	n/a
***SAV** <value>	n/a	<value> ::= {1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 }
:SEQuence:NEXT	n/a	n/a
:SEQuence:PREVious	n/a	n/a
:SEQuence:PROTect {ON | OFF}	:SEQuence:PROTect?	<protect> ::= {ON | OFF}, which reports the status of the protection.
:SEQuence:RESet	n/a	n/a

Command	Query	Options and Query Returns
:SEQuence:SETup {MASK | STEP}, <number>, <header> <setup_string>	:SEQuence:SETup? {MASK | STEP}, <number>	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, ::= #800000244. 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 <number>	: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** <mask>	*SRE?	<mask> ::= sum of all bits that are set, 0,...,255; an integer in NR1 format. <mask> ::= following values: Bit Weight Enables 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?** <display>	{ON | OFF} <display> ::= {CHANnel <n> | PMEMory{1 | 2}} for the 54600/01/02/03/15/16 {CHANnel <n> | PMEMory{1 | 2} | EXTernal} 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

Command	**Query**	**Options and Query Returns**
:STOP	n/a	n/a
:SYSTem:DSP <string>	n/a	<string> ::= quoted ASCII string
n/a	**:SYSTem:ERRor?**	<error> ::= an integer error code See error values in the online Programmer's Reference.
:SYSTem:KEY <key_code>	:SYSTem:KEY?	<key_code> ::= -1 to 16, or 19 to 50; an integer See key code values in the online Programmer's Reference.
:SYSTem:LOCK <value>	:SYSTem:LOCK?	<value> ::= {ON | OFF}
:SYSTem:SETup <setup_data>	:SYSTem:SETup?	<setup_data> ::= a maximum of 218 bytes of data in IEEE 488.2 # format.
n/a	**:TER?**	<return_value> ::= 0 or 1
:TIMebase:DELay <delay_value>	:TIMebase:DELay?	<delay_value> ::= time from trigger to display reference in seconds. The display reference is left or center in NR3 format.
:TIMebase:MODE <value>	:TIMebase:MODE?	<value> ::= {NORMal | DELayed | XY | ROLL}
:TIMebase:RANGe <range_value>	:TIMebase:RANGe?	<range_value> ::= the following values in NR3 format: 50 ns through 50 s for 54603 20 ns through 50 s for 54600/01/02 10 ns through 50 s for 54610/15/16
:TIMebase:REFerence {LEFT | CENTer}	:TIMebase:REFerence?	<return_value> ::= {LEFT | CENTer} for Normal or Delayed modes. <return_value> ::= {CENTer | RIGHt} for ROLL mode.
n/a	**:TIMebase:SETup?**	For all models except the 54615/16: TIMebase:MODE {NORM | DEL | XY};RANGe <range>; DELay <delay>;REF {LEFT | CENT};VERN {ON | OFF} For the 54615/16: TIMebase:MODE {NORM | DEL | XY};RANGe <range>; DELay <delay>;REF {LEFT | CENT} <range> ::= the following values in NR3 format: 50 ns through 50 s for 54603 20 ns through 50 s for 54600/01/02 10 ns through 50 s for 54610/15/16 <delay> ::= time from trigger to delay reference in seconds in NR3 format
:TIMebase:VERNier {ON | OFF}	:TIMebase:VERNier?	{ON | OFF}
:TRACe:CLEAR <N>	n/a	<N> ::= the trace memory number (1 to 100)
:TRACe:DATA <N>,<trace_data>	:TRACe:DATA? <N>	<N> ::= the trace memory number (1 to 100). <header> ::= the 10-byte block header that contains the ASCII characters #8000nnnnn and is sent prior to the data. (nnnnn 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.

Command	Query	Options and Query Returns
:TRACe:MODE <N> {ON | OFF}	:TRACe:MODE? <N>	<N> ::= 1 to 100 <return_state> ::= {ON | OFF}
:TRACe:SAVE <N>	n/a	<N> ::= the trace memory number (1 to 100).
***TRG**	n/a	n/a
:TRIGger:COUPling {AC | DC}	:TRIGger:COUPling?	{AC | DC}
:TRIGger:FIELd {ALTernate | ONE | TWO | VERTical}	:TRIGger:FIELd?	{ALTernate | ONE | TWO | VERTical}
:TRIGger:HOLDoff <holdoff_time>	:TRIGger:HOLDoff?	<holdoff_time> ::= the holdoff time value in seconds in NR3 format.
:TRIGger:LEVel <level_argument>	:TRIGger:LEVel?	<return_value> ::= the trigger level in volts in NR3 format.
:TRIGger:LINE <line_number>	:TRIGger:LINE?	<line_number> ::= integer in NR1 format.
:TRIGger:MODE {AUTLevel | AUTO | NORMal | SINGle | TV}	:TRIGger:MODE?	{AUTLevel | AUTO | NORMal | SINGle | TV}
:TRIGger:NREJect {OFF | ON}	:TRIGger:NREJect?	{OFF | ON}
:TRIGger:OPTMode {LINE | FIELD1 | FIELD2 | VERTical | ALLLINES | ALLFLDS}	:TRIGger:OPTMode?	{LINE | FIELD1 | FIELD2 | VERTical | ALLLINES | ALLFLDS}
:TRIGger:POLarity {POSitive | NEGative}	:TRIGger:POLarity?	{POS | NEG}
:TRIGger:REJect {OFF | LF | HF}	:TRIGger:REJect?	{OFF | LF | HF}
n/a	:TRIGger:SETup?	TRIG:MODE {AUTL | AUTO | NORM | SING | TV}; SOURCE <source>; LEVEL <level>; HOLD <time>; SLOPE {POS | NEG}; COUP {AC | DC}; REJ {OFF | LF | HF}; NREJ {ON | OFF}; POL {POS | NEG}; TVMODE <tvmode>; TVHF {ON | OFF} <level> ::= trigger level in volts in NR3 format <time> ::= holdoff time value in seconds in NR3 format <source> ::= {CHAN{1 | 2} | EXT | LINE} for 54600/03/10/15/16 {CHAN{1 | 2 | 3 | 4} | LINE} for 54601/02 <tvmode>::= {FIELD1 | FIELD2 | LINE} for 54600/01/03 {FIELD1 | FIELD2 | LINE | VERT} for 54602/10/15/16
:TRIGger:SLOPe {NEGative | POSitive}	:TRIGger:SLOPe?	{NEG | POS}

Command	Query	Options and Query Returns
:TRIGger:SOURce <source>	:TRIGger:SOURce?	<source> ::= {CHANnel1 | CHANnel2 | EXTernal | LINE} for 54600/03/10/15/16 {CHANnel1 | CHANnel2 | CHANnel3 | CHANnel4} for 54601/02
:TRIGger:STANdard {GENeric | NTSC | PAL | PALM | SECam}	:TRIGger:STANdard?	{GENeric | NTSC | PAL | SECam}
:TRIGger:TVHFrej {OFF | ON}	:TRIGger:TVHFrej?	{OFF | ON}
:TRIGger:TVMode <mode>	:TRIGger:TVMode?	<mode> ::= {LINE | FIELD1 | FIELD2} for 54600/01/03/15/16 {LINE | FIELD1 | FIELD2 | VERTical} for 54602/10
:TRIGger:VIR {ON | OFF}	:TRIGger:VIR?	{ON | OFF}
n/a	***TST?**	<result> ::= 0 or non-zero value; an integer in NR1 format 0 indicates the test passed. Non-zero indicates the test failed.
:VAUToscale	n/a	n/a
:VIEW <display>	n/a	<display> ::= {CHANnel <n> | PMEMory{1 | 2}} for the 54600/01/02/03/15/16 {CHANnel <n> | PMEMory{1 | 2} | EXTernal} 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
***WAI**	n/a	n/a
:WAVeform:BYTeorder <value>	:WAVeform:BYTeorder?	<value> ::= {LSBFirst | MSBFirst}
:WAVeform:DATA <binary block data in # format>	:WAVeform:DATA?	<binary block length bytes>, <binary data> For example, to transmit 4000 bytes of data, the syntax would be: #800004000<4000 bytes of data><NL> 8 is the number of digits that follow 00004000 is the number of bytes to be transmitted <4000 bytes of data> is the actual data
:WAVeform:FORMat <value>	:WAVeform:FORMat?	<value> ::= {ASC | WORD | BYTE}
:WAVeform:POINts <value>	:WAVeform:POINts?	<value> ::= integer {100 | 200 | 250 | 400 | 500 | 800 | 1000 | 2000 | 4000 | 5000} in NR1 format

Command	**Query**	**Options and Query Returns**
n/a	**:WAVeform:PREamble?**	<preamble_block> ::= <format NR1>, <type NR1>, <points NR1>, <count NR1>, <xincrement NR3>, <xorigin NR3>, <xreference NR1>, <yincrement NR3>, <yorigin NR3>, <yreference NR1> <format> ::= an integer in NR1 format: 0 for ASCii format 1 for BYTE format 2 for WORD format <type> ::= an integer in NR1 format: 0 for AVERage type 1 for NORMal type 2 for PEAK detect type <count> ::= an integer in NR1 format: 1, always 1 and is present for compatibility
:WAVeform:SOURce CHANnel <n>	:WAVeform:SOURce?	<n> ::= {1 | 2} for the 54600/03/10/15/16 {1 | 2 | 3 | 4} for the 54601/02
n/a	**:WAVeform:TYPE?**	<return_mode> ::= {NORMal | PEAK | AVERage}
n/a	**:WAVeform:XINCrement?**	<return_value> ::= x-increment in the current preamble in NR3 format
n/a	**:WAVeform:XORigin?**	<return_value> ::= x-origin value in the current preamble in NR3 format
n/a	**:WAVeform:XREFerence?**	<return_value> ::= x-reference value in the current preamble in NR1 format
n/a	**:WAVeform:YINCrement?**	<return_value> ::= y-increment value in the current preamble in NR3 format
n/a	**:WAVeform:YORigin?**	<return_value> ::= y-origin in the current preamble in NR3 format
n/a	**:WAVeform:YREFerence?**	<return_value>::= y-reference value in the current preamble in NR1 format

H.3 MUX

Agilent 34970A Data Acquisition/Switch Unit

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)
```

Agilent Technologies

Scan Measurement Commands

(*see page 226 in the User's Guide*)

```
S 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>)
   :FRESistance? [{<range>|AUTO|MIN|MAX|DEF}
      [,<resolution>|MIN|MAX|DEF}],] (@<scan_list>)
   :CURRent:DC? [{<range>|AUTO|MIN|MAX|DEF}
      [,<resolution>|MIN|MAX|DEF}],] (@<scan_list>)
   :CURRent:AC? [{<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>)
   :DIGital:BYTE? (@<scan_list>)
   :TOTalize? {READ|RRESet} ,(@<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>)]
```

S *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
S     :SCAN (@<scan_list>)
      :SCAN?
      :SCAN:SIZE?

G  TRIGger
      :SOURce {BUS|IMMediate|EXTernal|ALARm1|ALARm2|ALARm3|ALARm4|TIMer}
      :SOURce?

G  TRIGger
      :TIMer {<seconds>|MIN|MAX}
      :TIMer?

G  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>)]

G  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?

G  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_count>]
```

S *This command redefines the scan list when executed.*
G *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
S    :SCAN (@<scan_list>)
     :SCAN?
     :SCAN:SIZE?

G ROUTe
G TRIGger
     :SOURce {BUS|IMMediate|EXTernal|TIMer}
     :SOURce?

G TRIGger
     :TIMer {<seconds>|MIN|MAX}
     :TIMer?

G TRIGger
     :COUNt {<count>|MIN|MAX|INFinity}
     :COUNt?

  ROUTe
     :CHANnel:DELay <seconds>[,(@<ch_ list>)]
     :CHANnel:DELay? [(@<ch_list>)]

G ROUTe
     :CHANnel:ADVance:SOURce {EXTernal|BUS|IMMediate}
     :CHANnel:ADVance:SOURce?

  ROUTe
     :CHANnel:FWIRe {OFF|ON}[,(@<ch_list>)]
     :CHANnel:FWIRe? [(@<ch_list>)]

G INSTrument
     :DMM {OFF|ON}
     :DMM?
     :DMM:INSTalled?
```

S *This command redefines the scan list when executed.*
G *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*)

```
S 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:TRANsducer
   :TYPE {TCouple|RTD|FRTD|THERmistor|DEF}[,(@<ch_list>)]
   :TYPE? [(@<ch_list>)]

[SENSe:]TEMPerature:TRANsducer
   :TCouple:TYPE {B|E|J|K|N|R|S|T}[,(@<ch_list>)]
   :TCouple:TYPE? [(@<ch_list>)]
   :TCouple:CHECk {OFF|ON}[,(@<ch_list>)]
   :TCouple:CHECk? [(@<ch_list>)]

[SENSe:]TEMPerature:TRANsducer
   :TCouple:RJUNction:TYPE {INTernal|EXTernal|FIXed}[,(@<ch_list>)]
   :TCouple:RJUNction:TYPE? [(@<ch_list>)]
   :TCouple:RJUNction {<temperature>|MIN|MAX}[,(@<ch_list>)]
   :TCouple:RJUNction? [(@<ch_list>)]

[SENSe:]TEMPerature:RJUNction? [(@<ch_list>)]

[SENSe:]TEMPerature:TRANsducer
   :RTD:TYPE {85|91}[,(@<ch_list>)]
   :RTD:TYPE? [(@<ch_list>)]
   :RTD:RESistance[:REFerence] <reference>[,(@<ch_list>)]
   :RTD:RESistance[:REFerence]? [(@<ch_list>)]

[SENSe:]TEMPerature:TRANsducer
   :FRTD:TYPE {85|91}[,(@<ch_list>)]
   :FRTD:TYPE? [(@<ch_list>)]
   :FRTD:RESistance[:REFerence] <reference>[,(@<ch_list>)]
   :FRTD:RESistance[:REFerence]? [(@<ch_list>)]

[SENSe:]TEMPerature:TRANsducer
   :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}[,(@<ch_list>)]
   TEMPerature:NPLC? [{(@<ch_list>)|MIN|MAX}]
```

S *This command redefines the scan list when executed.*
Default parameters are shown in **bold**.

Voltage Configuration Commands

(*see page 223 in the User's Guide*)

```
S  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? [{(@<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? [{(@<ch_list>)|MIN|MAX}]

   [SENSe:]
     VOLTage:DC:APERture {<time>|MIN|MAX}[,(@<ch_list>)]
     VOLTage:DC:APERture? [{(@<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? [{(@<ch_list>)|MIN|MAX}]

   INPut
     :IMPedance:AUTO {OFF|ON}[,(@<ch_list>)]
     :IMPedance:AUTO? [(@<ch_list>)]

   [SENSe:]
     ZERO:AUTO {OFF|ONCE|ON}[,(@<ch_list>)]
     ZERO:AUTO? [(@<ch_list>)]

S  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? [{(@<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? [{(@<ch_list>)|MIN|MAX}]
```

S *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*)

```
S 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>)]

  [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>)]

S 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>)]

  [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>)]
```

S *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.

```
S CONFigure
    :CURRent:DC [{<range>|AUTO|MIN|MAX|DEF}
        [,<resolution>|MIN|MAX|DEF}],] (@<scan_list>)
  CONFigure? [(@<ch_list>)]

  [SENSe:]
    CURRent:DC:RANGe {<range>|MIN|MAX}[,(@<ch_list>)]
    CURRent:DC:RANGe? [{(@<ch_list>)|MIN|MAX}]
    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>)|MIN|MAX}]

  [SENSe:]
    CURRent:DC:APERture {<time>|MIN|MAX}[,(@<ch_list>)]
    CURRent:DC:APERture? [{(@<ch_list>)|MIN|MAX}]

  [SENSe:]
    CURRent:DC:NPLC {0.02|0.2|1|2|10|20|100|200|MIN|MAX}[,(@<ch_list>)]
    CURRent:DC:NPLC? [{(@<ch_list>)|MIN|MAX}]

S CONFigure
    :CURRent:AC [{<range>|AUTO|MIN|MAX|DEF}
        [,<resolution>|MIN|MAX|DEF}],] (@<scan_list>)
  CONFigure? [(@<ch_list>)]

  [SENSe:]
    CURRent:AC:RANGe {<range>|MIN|MAX}[,(@<ch_list>)]
    CURRent:AC:RANGe? [{(@<ch_list>)|MIN|MAX}]
    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>)|MIN|MAX}]
```

S *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*)

S
```
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}]
```

S
```
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}]
```

S *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>[,(@<ch_list>)]
  :SCALe:GAIN? [(@<ch_list>)]
  :SCALe:OFFSet <offset>[,(@<ch_list>)]
  :SCALe:OFFSet? [(@<ch_list>)]
  :SCALe:UNIT <quoted_string>[,(@<ch_ list>)]
  :SCALe:UNIT? [(@<ch_list>)]

CALCulate:SCALe:OFFSet:NULL [(@<ch_list>)]

CALCulate
  :SCALe:STATe {OFF|ON}[,(@<ch_list>)]
  :SCALe:STATe? [(@<ch_list>)]
```

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>[,(@<ch_ list>)]
  :LIMit:UPPer? [(@<ch_list>)]
  :LIMit:UPPer:STATe {OFF|ON}[,(@<ch_list>)]
  :LIMit:UPPer:STATe? [(@<ch_list>)]

CALCulate
  :LIMit:LOWer <lo_limit>[,(@<ch_ list>)]
  :LIMit:LOWer? [(@<ch_list>)]
  :LIMit:LOWer:STATe {OFF|ON}[,(@<ch_list>)]
  :LIMit:LOWer:STATe? [(@<ch_list>)]

SYSTem:ALARm?

G 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?
  :ALARm[:EVENt]?
```

Ch 01	Ch 02	Ch 03	Ch 04	Ch 05
DIO (LSB)	DIO (MSB)	Totalizer	DAC	DAC

```
CALCulate
  :COMPare:TYPE {EQUal|NEQual}[,(@<ch_list>)]
  :COMPare:TYPE? [(@<ch_list>)]
  :COMPare:DATA <data>[,(@<ch_list>)]
  :COMPare:DATA? [(@<ch_list>)]
  :COMPare:MASK <mask>[,(@<ch_list>)]
  :COMPare:MASK? [(@<ch_list>)]
  :COMPare:STATe {OFF|ON}[,(@<ch_list>)]
  :COMPare:STATe? [(@<ch_list>)]
```

G *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)

Ch 01 DIO (LSB)	Ch 02 DIO (MSB)	Ch 03 Totalizer	Ch 04 DAC	Ch 05 DAC

S
```
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)

Ch 01 DIO (LSB)	Ch 02 DIO (MSB)	Ch 03 Totalizer	Ch 04 DAC	Ch 05 DAC

S
```
CONFigure:TOTalize {READ|RRESet} ,(@<scan_list>)
CONFigure? [(@<ch_list>)]

[SENSe:]
  TOTalize:TYPE {READ|RRESet}[,(@<ch_list>)]
  TOTalize:TYPE? [(@<ch_list >)]

[SENSe:]
  TOTalize:SLOPe {NEGative|POSitive}[,(@<ch_list>)]
  TOTalize:SLOPe? [(@<ch_list>)]

[SENSe:]TOTalize:CLEar:IMMediate [(@<ch_ list>)]

[SENSe:]TOTalize:DATA? [(@<ch_ list>)]
```

Digital Output Commands

(see page 258 in the User's Guide)

Ch 01 DIO (LSB)	Ch 02 DIO (MSB)	Ch 03 Totalizer	Ch 04 DAC	Ch 05 DAC

```
SOURce
  :DIGital:DATA[:{BYTE|WORD}] <data> ,(@<ch_list>)
  :DIGital:DATA[:{BYTE|WORD}]? (@<ch _list>)

SOURce:DIGital:STATe? (@<ch_list>)
```

DAC Output Commands

(see page 258 in the User's Guide)

Ch 01 DIO (LSB)	Ch 02 DIO (MSB)	Ch 03 Totalizer	Ch 04 DAC	Ch 05 DAC

```
SOURce
  :VOLTage <voltage> ,(@<ch_list>)
  :VOLTage? (@<ch_list>)
```

S *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*)

```
G TRIGger
    :SOURce {BUS|IMMediate|EXTernal|ALARm1|ALARm2|ALARm3|ALARm4|TIMer}
    :SOURce?

G TRIGger
    :TIMer {<seconds>|MIN|MAX}
    :TIMer?

G 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?
```

G *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>
*SRE?

STATus
  :QUEStionable:CONDition?
  :QUEStionable[:EVENt]?
  :QUEStionable:ENABle <enable_value>
  :QUEStionable:ENABle?

*ESR?
*ESE <enable_value>
*ESE?

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?
```

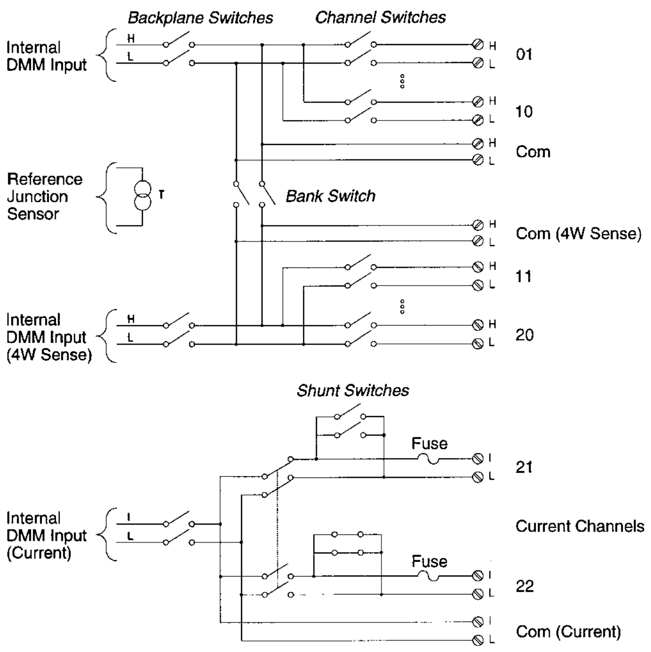
IEEE 488.2 Common Commands

```
*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**.

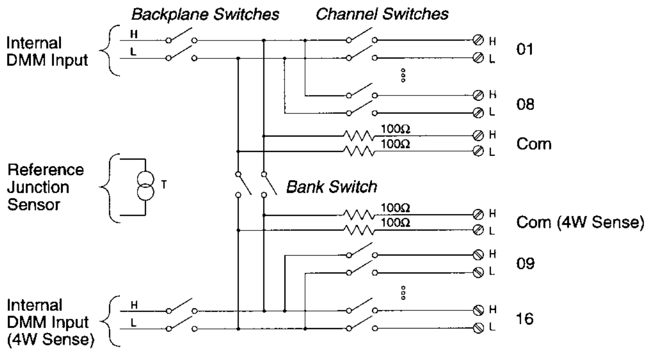
Agilent 34901A 20-Channel Multiplexer

(*see page 164 in the User's Guide*)



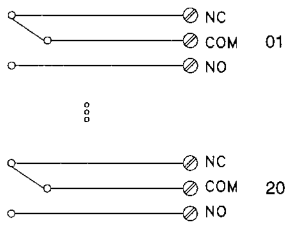
Agilent 34902A 16-Channel Multiplexer

(*see page 166 in the User's Guide*)



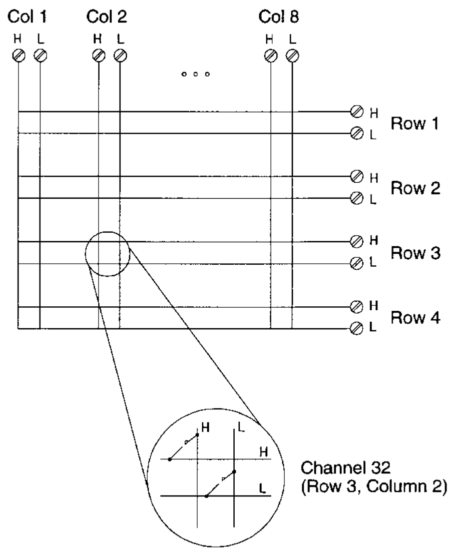
Agilent 34903A 20-Channel Actuator

(*see page 168 in the User's Guide*)



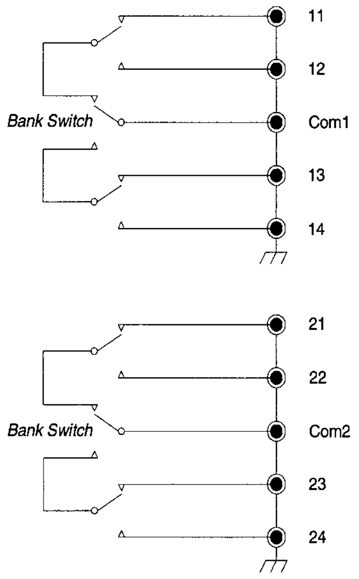
Agilent 34904A 4x8 Matrix

(*see page 170 in the User's Guide*)



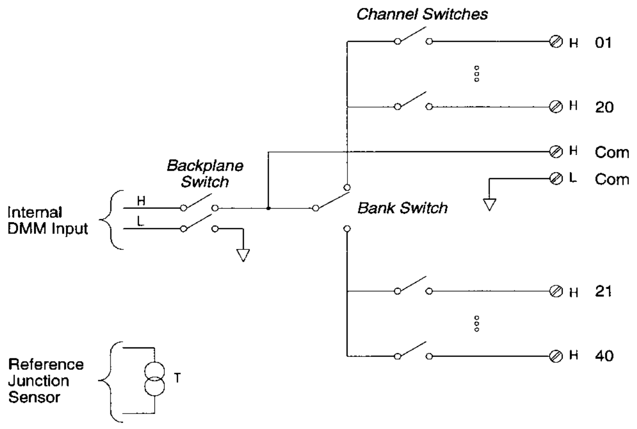
Agilent 34905A/6A Dual 4-Channel RF Multiplexers

(*see page 172 in the User's Guide*)



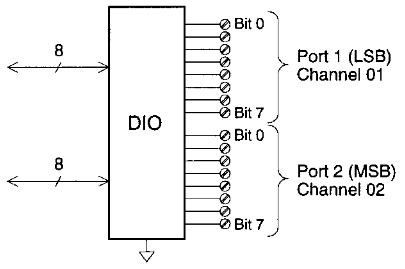
Agilent 34908A 40-Channel Single-Ended Multiplexer

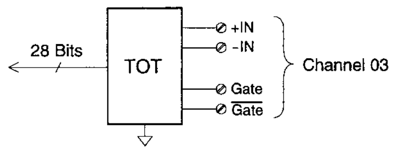
(*see page 174 in the User's Guide*)

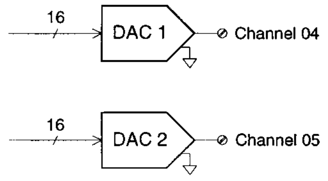


Agilent 34907A Multifunction Module

(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.

Measurement Configuration	**Factory Reset State**
Function	DC Volts
Range	Autorange
Resolution	5½ digits
Integration Time	1 PLC
Input Resistance	10 MΩ (fixed for all DCV ranges)
Channel Delay	Automatic Delay
Totalizer Reset Mode	Count Not Reset When Read
Totalizer Edge Detect	Rising Edge
Scanning Operations	**Factory Reset State**
Scan List	Empty
Reading Memory	All Readings are Cleared
Min, Max, and Average	All Statistical Data is Cleared
Scan Interval Source	Immediate
Scan Interval	Front Panel = 10 Seconds Remote = Immediate
Scan Count	Front Panel = Continuous Remote = 1 Scan Sweep
Scan Reading Format	Reading Only (No Units, Channel, Time)
Monitor in Progress	Stopped
Mx+B Scaling	**Factory Reset State**
Gain Factor ("M")	1
Scale Factor ("B")	0
Scale Label	Vdc
Alarm Limits	**Factory Reset State**
Alarm Queue	Not Cleared
Alarm State	Off
HI and LO Alarm Limits	0
Alarm Output	Alarm 1
Alarm Output Configuration	Latched Mode
Alarm Output State	Output Lines are Cleared
Alarm Output Slope	Fail = Low
Module Hardware	**Factory Reset State**
34901A, 34902A, 34908A	Reset: All Channels Open
34903A, 34904A	Reset: All Channels Open
34905A, 34906A	Reset: Channels **s**11 and **s**21 Selected
34907A	Reset: Both DIO Ports = Input, Count = 0, Both DACs = 0 Vdc
System-Related Operations	**Factory Reset State**
Display State	On
Error Queue	Errors Not Cleared
Stored States	No Change

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

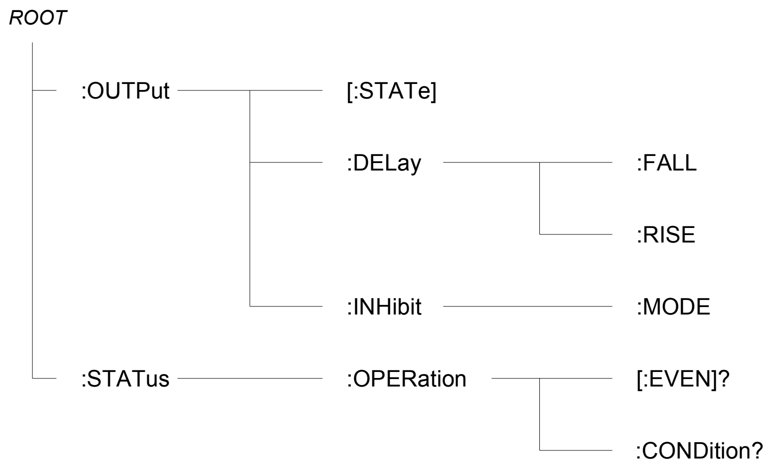
This chapter contains a brief introduction to the SCPI Programming language. SCPI (Standard Commands for Programmable Instruments) is a programming language for controlling instrument functions over the GPIB.

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>
```

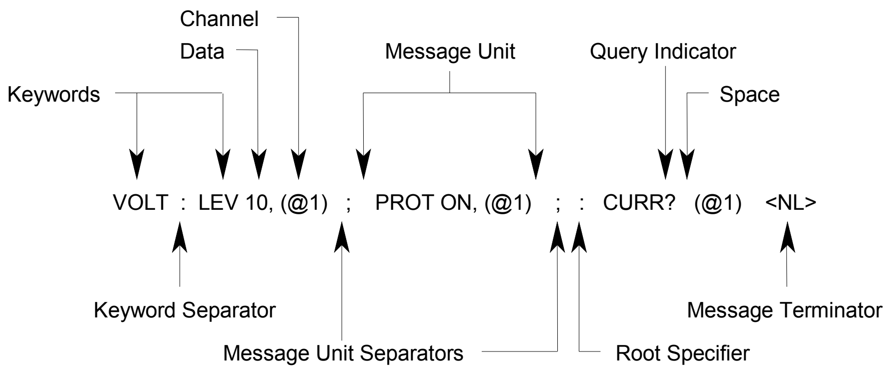
will always be correct because the commands are not executed until the message terminator is received. Because the range and setting information is received as a set, no range/setting conflict occurs.

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.

```
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.

Angle brackets < >	Items within angle brackets are parameter abbreviations. For example, <NR1> indicates a specific form of numerical data.
Vertical bar |	Vertical bars separate alternative parameters. For example, VOLT | CURR indicates that either "VOLT" or "CURR" can be used as a parameter.
Square Brackets [ ]	Items within square brackets are optional. The representation [SOURce:]VOLTage means that SOURce: may be omitted.
Braces { }	Braces indicate parameters that may be repeated zero or more times. It is used especially for showing arrays. The notation <A>{<,B>} shows that parameter "A" must be entered, while parameter "B" may be omitted or may be entered one or more times.
Parentheses ( )	Items within parentheses are used in place of the usual parameter types to specify a channel list. The notation (@1:3) specifies a channel list that includes channels 1, 2, and 3. The notation (@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

Symbol	Response Formats
<NR1>	Digits with an implied decimal point assumed at the right of the least-significant digit. Examples: 273
<NR2>	Digits with an explicit decimal point. Example: .0273
<NR3>	Digits with an explicit decimal point and an exponent. Example: 2.73E+2
	Parameter Formats
<NRf>	Extended format that includes <NR1>, <NR2> and <NR3>. Examples: 273 273. 2.73E2
<NRf+>	Expanded decimal format that includes <NRf> 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.
<Bool>	Boolean Data. They can be numeric (0, 1), or named (ON, OFF).
<SPD>	String program data. String parameters enclosed in single or double quotes.

Suffixes and Multipliers

Class	Suffix	Unit	Unit with Multiplier
Current	A	ampere	MA (milliampere)
Amplitude	V	volt	MV (millivolt)
Time	S	second	MS (millisecond)
Common Multipliers			
1E3	K	kilo	
1E-3	M	milli	
1E-6	U	micro	

Response Data Types

Symbol	Response Formats
<CRD>	Character Response Data. Permits the return of character strings.
<AARD>	Arbitrary ASCII Response Data. Permits the return of undelimited 7-bit ASCII. This data type has an implied message terminator.
<SRD>	String Response Data. Returns string parameters enclosed in single or double quotes.

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
```

6
Language Dictionary

SCPI Command Summary 76

Calibration Subsystem 80

Measurement Subsystem 82

Output Subsystem 86

Source Subsystem 89

Status Subsystem 96

System Commands 103

Trigger Subsystem 107

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.

SCPI Command Summary

Subsystem Commands

NOTE Some [optional] commands have been included for clarity.
Not all commands apply to all models.

SCPI Command	Description
ABORt	
:ACQuire (@chanlist)	Resets the measurement trigger system to the Idle state
:TRANsient (@chanlist)	Resets the transient trigger system to the Idle state
CALibrate	
:CURRent	
[:LEVel] <NRf>, (@channel)	Calibrates the output current programming
:MEASure <NRf>, (@channel)	Calibrates the current measurement
:PEAK (@channel)	Calibrates the peak current limit (Agilent N6751A/52A/61A/62A)
:DATA <NRf>	Enters the calibration value
:DATE <SPD>, (@channel)	Sets the calibration date
:DPRog (@channel)	Calibrates the current downprogrammer
:LEVel P1 | P2 | P3	Advances to the next calibration step
:PASSword <NRf>	Sets the numeric calibration password
:SAVE	Saves the new cal constants in non-volatile memory
:STATE <Bool> [,<NRf>]	Enables/disables calibration mode
:VOLTage	
[:LEVel] <NRf>, (@channel)	Calibrates the output voltage programming
:CMRR (@channel)	Calibrates common mode rejection ratio (N6751A/52A/61A/62A)
:MEASure <NRf>, (@channel)	Calibrates the voltage measurement
DISPlay[:WINDow]:VIEW METER1 | METER4	Selects 1-channel or 4-channel meter view
FETCh (Note 1) | MEASure	
[:SCALar]	
:CURRent [:DC]? (@chanlist)	Returns the average output current
:VOLTage [:DC]? (@chanlist)	Returns the average output voltage
:ARRay	(Array commands only on Agilent N6761A/62A and Option 054)
:CURRent [:DC]? (@chanlist)	Returns the instantaneous output current
:VOLTage [:DC]? (@chanlist)	Returns the instantaneous output voltage
INITiate	
[:IMMediate]	(Acquire command only on Agilent N6761A/62A and Option 054)
:ACQuire (@chanlist)	Enables the measurement system to receive triggers
:TRANsient (@chanlist)	Enables the output transient system to receive triggers
:CONTinuous	
:TRANsient <Bool>, (@chanlist)	Enables/disables continuous transient triggers
OUTPut	
[:STATe] <Bool>, [NORelay], (@chanlist)	Enables/disables the specified output channel(s)
:DELay	
:FALL <NRf+>, (@chanlist)	Sets the output turn-off sequence delay
:RISE <NRf+>, (@chanlist)	Sets the output turn-on sequence delay
:INHibit	
:MODE LATChing | LIVE | OFF	Sets the remote inhibit input

SCPI Command	Description
OUTPut (continued)	
:PON	
:STATe RST | RCL0	Programs the power-on state
:PROTection	
:CLEar (@chanlist)	Resets latched protection
:COUPle <Bool>	Enables/disables channel coupling for protection faults
:DELay <NRf+>, (@chanlist)	Sets over-current protection programming delay
SENSe	
:CURRent [:DC]	
:RANGe [:UPPer] <NRf+>, (@chanlist)	Selects the current measurement range (Agilent N6761A/62A)
:FUNCtion "VOLTage" | "CURRent", (@chanlist)	Selects the measurement function
:SWEep	(Sweep commands only on Agilent N6761A/62A and Option 054)
:OFFSet:POINts <NRf+>, (@chanlist)	Defines the trigger offset in the measurement sweep
:POINts <NRf+>, (@chanlist)	Defines the number of data points in the measurement
:TINTerval <NRf+>, (@chanlist)	Sets the measurement sample interval
:VOLTage [:DC]	
:RANGe [:UPPer] <NRf+>, (@chanlist)	Selects the voltage measurement range (Agilent N6761A/62A)
:WINDow [:TYPE] HANNing | RECTangular, (@chanlist)	Selects the measurement window (N6761A/62A and Option 054)
[SOURce:]	
CURRent	
[:LEVel]	
[:IMMediate][:AMPLitude] <NRf+>, (@chanlist)	Sets the output current
:TRIGgered [:AMPLitude] <NRf+>, (@chanlist)	Sets the triggered output current
:MODE FIXed | STEP | LIST, (@chanlist)	Sets the current trigger mode
:PROTection	
:STATe <Bool>, (@chanlist)	Enables/disables over-current protection on the selected output
:RANGe <NRf+>, (@chanlist)	Sets the output current range (Agilent N6761A/62A)
DIGital	
:INPut:DATA?	Reads the state of the digital port pins
:OUTPut:DATA <NRf>	Sets the digital port
:PIN1 | :PIN2 | :PIN3 | :PIN4 | :PIN5 | :PIN6 | :PIN7	
:FUNCtion DIO | DINP | TOUT | TINP | FAUL[1] | INH[2]	Sets the selected pin's function ([1]PIN1 only; [2]PIN3 only)
:POLarity POSitive | NEGative	Sets the selected pin's polarity
LIST	(List commands only on Agilent N6761A/62A and Option 054)
:COUNt <NRf+> | INFinity, (@chanlist)	Sets the list repeat count
:CURRent [:LEVel] <NRf> {,<NRf>. . .}, (@chanlist)	Sets the current list
:POINts? (@chanlist)	Returns the number of current list points
:DWELl <NRf> {,<NRf>. . .}, (@chanlist)	Sets the list of dwell times
:POINts? (@chanlist)	Returns the number of dwell list points
:STEP ONCE | AUTO, (@chanlist)	Specifies how the list responds to triggers
:TERMinate	
:LAST <Bool>, (@chanlist)	Sets the list termination mode
:TOUTput	
:BOSTep[:DATA] <Bool> {,<Bool>. . .}, (@chanlist)	Sets the steps to generate triggers at the Begin Of Step
:POINts? (@chanlist)	Returns the number of beginning of step list points
:EOSTep[:DATA] <Bool> {,<Bool>. . .}, (@chanlist)	Sets the steps to generate triggers at the End Of Step
:POINts? (@chanlist)	Returns the number of end of step list points
:VOLTage[:LEVel] <NRf> {,<NRf>. . .}, (@chanlist)	Sets the voltage list
:POINts? (@chanlist)	Returns the number of voltage level points
STEP	
:TOUTput <Bool>, (@chanlist)	Generate a trigger output on the voltage or current step transient

SCPI Command	Description
[SOURce:] (continued)	
VOLTage	
[:LEVel]	
[:IMMediate][:AMPLitude] <NRf+>, (@chanlist)	Sets the output voltage
:TRIGgered [:AMPLitude] <NRf+>, (@chanlist)	Sets the triggered output voltage
:MODE FIXed | STEP | LIST, (@chanlist)	Sets the voltage trigger mode
:PROTection	
[:LEVel] <NRf+>, (@chanlist)	Sets the over-voltage protection level
:RANGe <NRf+>, (@chanlist)	Sets the output voltage range (Agilent N6761A/62A)
:SLEW <NRf+> | INFinity, (@chanlist)	Sets the output voltage slew rate
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
:NTRansition <NRf>, (@chanlist)	Sets the Negative transition filter
:PTRansition <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
:NTRansition <NRf>, (@chanlist)	Sets the Negative transition filter
:PTRansition <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 | REMote | RWLock	Specifies the Remote/Local state of the instrument
: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)
[:IMMediate] (@chanlist)	Triggers the measurement immediately
:SOURce BUS | PIN<pin> | TRAN<chan>, (@chanlist)	Sets the measurement trigger source
:TRANsient	
[:IMMediate] (@chanlist)	Triggers the output immediately
:SOURce BUS | PIN<pin> | TRAN<chan>, (@chanlist)	Sets the output trigger source

Common Commands

Command	Description	Command	Description
*CLS	Clear status	*RDT?	Return output channel descriptions
*ESE <NRf>	Standard event status enable	*RST	Reset
*ESE?	Return standard event status enable	*SAV <NRf>	Saves an instrument state
*ESR?	Return event status register	*SRE <NRf>	Set service request enable register
*IDN?	Return instrument identification	*SRE?	Return service request enable register
*OPC	Enable "operation complete" bit in ESR	*STB?	Return status byte
*OPC?	Return a "1" when operation complete	*TRG	Trigger
*OPT?	Return option number	*TST?	Performs self-test, then returns result
*RCL <NRf>	Recalls a saved instrument state	*WAI	Pauses additional command processing until all device commands are done

*RST Settings

These settings are set by the *RST (Reset) command

Calibration Function (Note 1)		**Measurement (continued)**	
CAL:STAT	OFF	SENS:SWE:OFFS:POIN	0
Current Function		SENS:SWE:TINT	20.48E–6
[SOUR:]CURR	80 mA	SENS:VOLT:RANG	MAX
[SOUR:]CURR:MODE	FIX	SENS:WIND	RECT
[SOUR:]CURR:PROT:STAT	OFF	**Output Function**	
[SOUR:]CURR:RANG	MAX	OUTP	OFF
[SOUR:]CURR:TRIG	MIN	OUTP:DEL:FALL	0
Digital Function		OUTP:DEL:RISE	0
[SOUR:]DIG:OUTP:DATA	0	OUTP:PROT:COUP	OFF
Display Function		OUTP:PROT:DEL	0.02
DISP:VIEW	METER1	OUTP:REL	OFF
List Function (Note 1)		**Step Function**	
[SOUR:]LIST:COUN	1	[SOUR:]STEP:TOUT	FALSE
[SOUR:]LIST:CURR	MIN	**Trigger Function**	
[SOUR:]LIST:DWEL	0.001	INIT:CONT:TRAN	OFF
[SOUR:]LIST:STEP	AUTO	TRIG:ACQ:SOUR	BUS
[SOUR:]LIST:TERM:LAST	OFF	TRIG:TRAN:SOUR	BUS
[SOUR:]LIST:TOUT:BOST	OFF	**Voltage Function**	
[SOUR:]LIST:TOUT:EOST	OFF	[SOUR:]VOLT	MIN
[SOUR:]LIST:VOLT	MIN	[SOUR:]VOLT:MODE	FIX
Measurement Function		[SOUR:]VOLT:PROT:LEV	MAX
SENS:CURR:RANG	MAX	[SOUR:]VOLT:RANG	MAX
SENS:FUNC	“VOLT”	[SOUR:]VOLT:SLEW	MAX
SENS:SWE:POIN	1024	[SOUR:]VOLT:TRIG	MIN

Note 1 The calibration state and all list settings are not saved by the *SAV command.

Calibration Subsystem

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. **P1** is the first calibration level; **P2** is the second level; **P3** is the third level.

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 CALibration:STATe OFF without first saving the new constants, the previous constants are restored.

CALibrate:STATe {ON|OFF} [,<password>]
CALibrate:STATe?

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.

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.

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.

[SOURce:]CURRent:RANGe {<value>|MIN|MAX}, (@<chanlist>)
[SOURce:]CURRent: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 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 [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.

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:]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.

[SOURce:]DIGital:OUTPut:DATA <value>
[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.

Pin	Bit	Pin	Bit
1	0	4	3
2	1	5	4
3	2	6	5
		7	6

The query returns the last programmed value of the bits. To read the actual state of the pin, use [SOURce:]DIGital:INPut:DATA?

[SOURce:]DIGital:PIN1:FUNCtion {DIO|DINPut|TOUTput|TINPut|FAULt}
[SOURce:]DIGital:PIN2:FUNCtion {DIO|DINPut|TOUTput|TINPut}
[SOURce:]DIGital:PIN3:FUNCtion {DIO|DINPut|TOUTput|TINPut|INHibit}
[SOURce:]DIGital:PIN4:FUNCtion {DIO|DINPut|TOUTput|TINPut}
[SOURce:]DIGital:PIN5:FUNCtion {DIO|DINPut|TOUTput|TINPut}
[SOURce:]DIGital:PIN6:FUNCtion {DIO|DINPut|TOUTput|TINPut}
[SOURce:]DIGital:PIN7:FUNCtion {DIO|DINPut|TOUTput|TINPut}
[SOURce:]DIGital:PIN<1-7>:FUNCtion?

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.

[SOURce:]DIGital:PIN1:POLarity {POSitive|NEGative}
[SOURce:]DIGital:PIN2:POLarity {POSitive|NEGative}
[SOURce:]DIGital:PIN3:POLarity {POSitive|NEGative}
[SOURce:]DIGital:PIN4:POLarity {POSitive|NEGative}
[SOURce:]DIGital:PIN5:POLarity {POSitive|NEGative}
[SOURce:]DIGital:PIN6:POLarity {POSitive|NEGative}
[SOURce:]DIGital:PIN7:POLarity {POSitive|NEGative}
[SOURce:]DIGital:PIN<1-7>:POLarity?

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.

[SOURce:]LIST:COUNt {<count>|MIN|MAX|INFinity}, (@<chanlist>)
[SOURce:]LIST:COUNt? (@<chanlist>)

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.

[SOURce:]LIST:CURRent[:LEVel] <curr> {,<curr>}, (@<chanlist>)
[SOURce:]LIST:CURRent[:LEVel]? (@<chanlist>)

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.

[SOURce:]LIST:CURRent:POINts? (@<chanlist>)

This query returns the number of points (steps) programmed in the current list. Applies only to models with list capability.

[SOURce:]LIST:DWELl <time> {,<time>}, (@<chanlist>)
[SOURce:]LIST:DWELl? (@<chanlist>)

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:

Range in seconds	Resolution
0 to 0.262143	1 microsecond
0 to 2.62143	10 microseconds
0 to 26.2143	100 microseconds
0 to 262.143	1 millisecond

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.

[SOURce:]LIST:DWELl:POINts? (@<chanlist>)

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.*

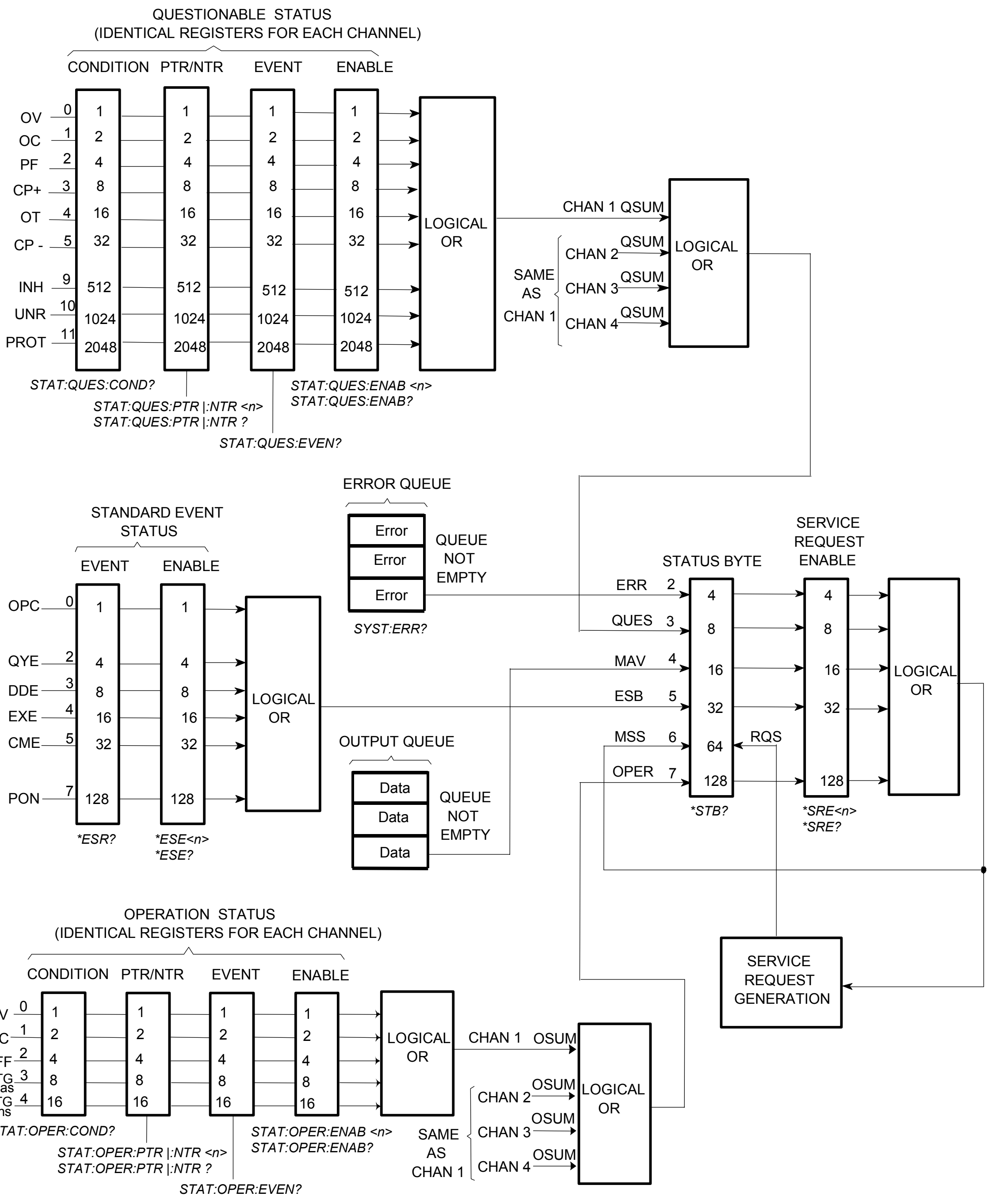
MSS and RQS Bits

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.

Operation Register	Questionable Register	Preset Settings
STAT:OPER:ENAB	STAT:QUES:ENAB	0 all bits disabled
STAT:OPER:NTR	STAT:QUES:NTR	0 all bits disabled
STAT:OPER:PTR		31 all defined bits enabled
	STAT:QUES:PTR	3647 all defined bits enabled

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:

Bit Position	15-5	4	3	2	1	0
Bit Value	–	16	8	4	2	1
Bit Name	–	WTG-tran	WTG-meas	OFF	CC	CV

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>)
STATus:OPERation:ENABle? (@<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:

Bit Position	15-12	11	10	9	8-6	5	4	3	2	1	0
Bit Value	–	2048	1024	512	–	32	16	8	4	2	1
Bit Name	–	PROT	UNR	INH	–	CP –	OT	CP+	PF	OC	OV

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:PTRansiton <value>, (@<chanlist>)
STATus:QUEStionable:NTRansition? (@<chanlist>)
STATus:QUEStionable:PTRansition? (@<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
*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:

Bit Position	7	6	5	4	3	2	1	0
Bit Value	128	–	32	16	8	4	–	1
Bit Name	PON	–	CME	EXE	DDE	QUE	–	OPC

PON = Power-on has occurred	DDE = Device-dependent error
CME = Command error	QUE = Query error
EXE = Execution 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
*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
*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.

Bit Position	7	6	5	4	3	2	1 – 0
Bit Value	128	64	32	16	8	4	–
Bit Name	OPER	MSS (RQS)	ESB	MAV	QUES	ERR	–

OPER = Operation status summary	MAV = Message available
MSS = Master status summary	QUES = Questionable status summary
(RQS) = Request for service	ERR = Error queue not empty
ESB = Event status byte summary	

*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: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.

Field	**Information**
Agilent Technologies	Manufacturer
N67xxA	Mainframe model number followed by a letter suffix
0	Zero or mainframe serial number if available
<A>.xx.xx	Revision levels of firmware

`*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.

*RDT?

This query returns a description of the output channels installed in a mainframe. Semicolons separate multiple channel descriptions.

Field	Information
CHAN <c>	Channel number
description	Description of the output channel

*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:

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:TOUTput, [SOURce:]LIST:TOUTput:BOSTep, and [SOURce:]LIST:TOUTput: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>)`
`TRIGger:TRANsient:SOURce?`

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:TOUTput, [SOURce:]LIST:TOUTput:BOSTep, and [SOURce:]LIST:TOUTput: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.

7
Programming Examples

Output Programming Example 110

List Programming Example 111

Digitizer Programming Example 113

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.

NOTE

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.

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.

```
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::US00000002"

    ' 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
        ' 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="TCPIP0::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 timeInterval 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::US00000002"

    ' 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"
        ' 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
    End With

End Sub
```