Automate Your VNA Application

Here is a thorough tutorial review of instrument control using the IEEE-488/GPIB bus, using a vector network analyzer as an example

By Jay Michael
National Instruments

Test and measurement facilities often have several desktop systems that are difficult to set up and control. With the GPIB bus, you can control multiple instruments with just one hardware interface in your PC. Automating such an application involves programming of instrumentation commands, configuration, message passing and so on. There are several factors to consider in automating a Vector Network Analyzer application, including the concept and building as well as the advantages of instrument drivers.

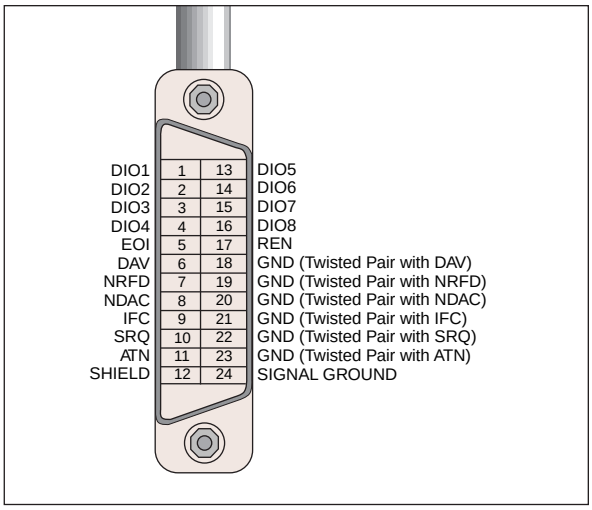
Vector Network Anlyzer (VNA) application

Instruments are traditionally controlled from their front panels. Getting the phase magnitude plots on a VNA for a simple two port network might take a minimum of four to six key punches. It may be easy to set the voltage level of a power supply or to take an accurate reading on a multi-meter with just a few button pushes. However, in the case of assembly line work and other high-productivity settings, controlling your instrument in this way is both time consuming and inefficient. The prospect of controlling three or four instruments for a single test multiplies such a daunting task.

With the advent of the GPIB, engineers and scientists have been automating instruments to perform iterative tasks with relative ease. In order to program these instruments, there are some basic concepts that need to be understood.

GPIB basics (Protocol, handshaking, commands)

GPIB, or the General Purpose Interface Bus, is an 8-bit parallel communications interface with data transfer rates up to 1 Mb/s. In 1975, the Institute of Electrical and Electronic Engineers (IEEE) published ANSI/IEEE Standard 488-1975, IEEE Standard Digital Interface for Programmable Instrumentation, which contained the electrical, mechanical and functional specifications of an instrument interfacing system.



■ **Figure 1. GPIB connector and pin assignments.**

The original IEEE 488 document contained no guidelines for a preferred syntax and format convention. This resulted in a supplement standard, IEEE 488.2, Codes, Formats, Protocols and Common Commands, for use with IEEE 488 (which was renamed IEEE 488.1). IEEE 488.2 does not replace IEEE 488 and many devices still conform only to IEEE 488.1. IEEE 488.2 builds on IEEE 488.1 by defining a minimum set of device interface capabilities, a common set of data codes and formats, a device message protocol, a generic set of commonly needed device commands and a new status reporting model. The bus supports one system controller, usually a computer, and up to fourteen addition-

al instruments. The GPIB has 16 signal lines and eight ground return or shield drain lines (see Figure 1). All GPIB devices share the same 24 bus lines. The 16 signal lines fall into the following groups: a) eight data lines; b) five interface management lines; and c) three handshake lines.

The eight data lines, DIO1 through DIO8, carry all the command and data messages on the GPIB. All commands and most data use the 7-bit ASCII or ISO code set, in which case the eighth bit, DIO8, is used for parity or not used at all. The interface management lines manage the flow of information across the GPIB. They are Interface Clear (IFC), Attention (ATN), Remote Enable (REN), End or Identify (EOI) and Service Request (SRQ). Three handshake lines asynchronously control the transfer of message bytes among devices: Not Ready For Data (NRFD), Not Data Accepted (NDAC), and Data Valid (DAV). The GPIB uses a three-wire interlocking handshake scheme. This scheme guarantees that devices send and receive message bytes on the data lines without transmission error [1].

To achieve the GPIB's high data transfer rate, you must limit the physical distance between devices and the number of devices on the bus, because the GPIB is a transmission line system. Any distance beyond the maximum allowable cable length, as well as any excess GPIB device loads, may surpass interface circuit drive capability. The IEEE 488 specification dictates a maximum separation of 4 m between any two devices and an average separation of 2 m over the entire bus. For example, if two devices are connected to the computer using the GPIB, and one device is 3 m away from the computer, then the second device should be 1 m from either the computer or the first device to give an average of 2 m over the entire bus. Other limitations include a maximum cable length of 20 m. A maximum of 15 devices (14 devices plus 1 controller) can be connected to each bus, with at least two-thirds of the devices powered on.

Instrument commands and data flow

You can communicate command messages to the GPIB or the instrument using the standard IEEE-488.1 command set. These commands are classified as Device Level Functions and Board Level Functions, depending on how you addressed the board. Device-level communication means you originally specified communication with a device on the bus. In board-level communication, you configure communication with the GPIB interface board rather than an instrument. When GPIB communication is at board level, you must manually configure the bus for all communication. Some of the board level commands include:

ibfind	Open and initialize a GPIB board
ibrsc	Request or release system control
ibrsv	Request service and change the serial poll status byte
ibsic	Assert interface clear
ibsre	Set or clear the Remote Enable (REN) line

Some of the device level commands include:

ibdev	Open and initialize a device
ibclr	Clear a specific device
ibconfig	Change the software configuration parameters
ibwrt	Write data to a device from a user buffer

Passing through the GPIB occurs via the same eight data lines used to pass commands. The bus management line, ATN, determines what type of message you are sending on the bus. If this line is unasserted, the information on the bus is a *data message*; if this line is asserted, the information is a *command message*. The devices on the GPIB monitor the ATN line, determine the data type, and treat the data appropriately.

Configuration (High speed timing, addressing, Controller-In-Charge, EOI)

To ensure that data passes reliably and that instruments do not use the bus simultaneously, set devices to be either Controllers, Talkers or Listeners. When two devices communicate, one will be a Talker and the other will be a Listener. In addition, one device on the bus must always be a Controller.

Most GPIB systems consist of one computer and a variety of instruments. In this type of system, the computer is typically the System Controller. If you have multiple computers, several devices can have controller capability, though only one Controller at a time is active or makes itself Controller-In-Charge (CIC). Active control can pass from the current CIC to an idle Controller. You can define the System Controller through jumper settings on the GPIB interface board, a software configuration file, or both.

The Controller defines communication links, responds to devices requesting service, sends GPIB commands and passes/receives control. Each device on the GPIB accepts its own command set and has its own method of terminating data strings. They are configured as either Talkers or Listeners, depending on what the controller instructs them to do. Once instructed, Talkers place data on the GPIB and Listeners read data that the Talker places on the GPIB. The Controller permits only one device at a time to talk, although several devices can be Listeners at the same time. Hence, the PC itself is now a Controller, a Talker and a Listener. Before any communication can take place on the bus, you must address the Talker and Listener. Before any data is passed between devices, the Controller first determines which will talk and which will listen.

Before you can talk to an instrument you need to make sure it has been addressed correctly. On the GPIB, each device (including the Controller) has a unique GPIB address. The primary GPIB address is in the range of decimal 0 to 30.

GPIB devices can also have secondary addresses, as described later in this section. The Controller sends a single byte (8 bits) of information for a Talker or Listener address command message. The Address com-

mand message has the following format: Bits 0 through 4 contain the binary GPIB primary address of the device in communication, and either bit 5, Listener Address (LA), or bit 6, Talker Address (TA), will be set if the device is a Talker or a Listener. Bit 7 is never used and is considered a "don't care" bit.

A device can also have a secondary address. A secondary address is in the range Hex 60 through Hex 7E. Address a device with a secondary address by sending the primary GPIB address and then the corresponding secondary address.

Once you have specified the address of the instrument and the data has been sent, a specifier is necessary to recognize the end of data transfer. Depending on the instrument and the type of data transfer, you can use three different methods: EOS; EOI; and the count method.

The *EOS method* uses an End of String (EOS) character, which signifies the termination of data that devices send on the GPIB. The Listener reads individual data bytes from the Talker until the EOS character is read. It then completes the read operation.

The *EOI method*, the most common method, uses the GPIB EOI (End or Identify) line. The EOI line is separate from the eight data lines on the GPIB. When the Talker sends the last byte of data in the transmission, the Talker sets the EOI line high to specify that the byte is the last to be sent. The Listener monitors the EOI line and recognizes that there is no more data. You must establish ahead of time whether the Talker will use the EOI method, so you can correctly configure the Listener to watch the EOI line.

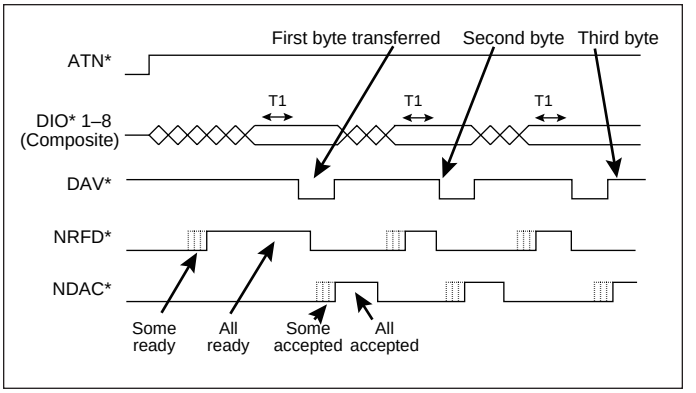
In using the *count method*, specify the count of the bytes to read, and reading terminates as soon as the count is reached.

Message passing (data format, handshaking, IEEE 488.2 issues, status reporting)

GPIB devices communicate with other GPIB devices by sending device-dependent messages and interface messages through the interface system. Device-dependent messages, often called data or data messages, contain device-specific information, such as programming instructions, measurement results, machine status and data files. Interface messages manage the bus. Usually called command messages, interface messages perform such functions as initializing the bus, addressing and unaddressing devices, and setting device modes for remote or local programming.

The eight data lines, DIO1 through DIO8, carry both data and command messages. The state of the Attention (ATN) line determines whether the information is data or commands. All commands and most data use the 7-bit ASCII or ISO code set, in which case the eighth bit, DIO8, is either unused or used for parity.

Three lines asynchronously control the transfer of message bytes between devices. The process is called a 3-wire interlocked handshake. It guarantees that message bytes on the data lines are sent and received without transmission error. NRFD (Not Ready For Data) indicates when a device is ready or not ready to receive a message byte. The line is driven by all devices when receiving commands, by Listeners when receiving data messages, and by the Talker when enabling the HS4881 protocol. NDAC (Not Data Accepted) indicates when a device has or has not accepted a message byte. DAV (Data Valid) indicates when the signals on the data lines are stable (valid) and can be accepted safely by devices. The Controller drives DAV when sending commands, and the Talker drives DAV when sending data messages.



■ **Figure 2. Normal IEEE 488.1 handshake.**

The standard IEEE 488-1975 3-wire handshake shown in Figure 2 requires the Listener to unassert Not Ready for Data (NRFD), the Talker to assert the Data Valid (DAV) signal to indicate to the Listener that a data byte is available, and the Listener to unassert the Not Data Accepted (NDAC) signal when it has accepted that byte. A byte cannot transfer in less time than it takes for the following events to occur — NRFD propagates to the Talker; DAV signal propagates to all Listeners; Listeners accept the byte and assert NDAC; NDAC propagates back to the Talker; Talker allows time for settling (T1) before asserting DAV again.

Five lines manage the flow of information across the interface:

ATN (Attention) — The Controller drives ATN (attention) true when it uses the data lines to send commands, and drives ATN false when a Talker can send data messages.

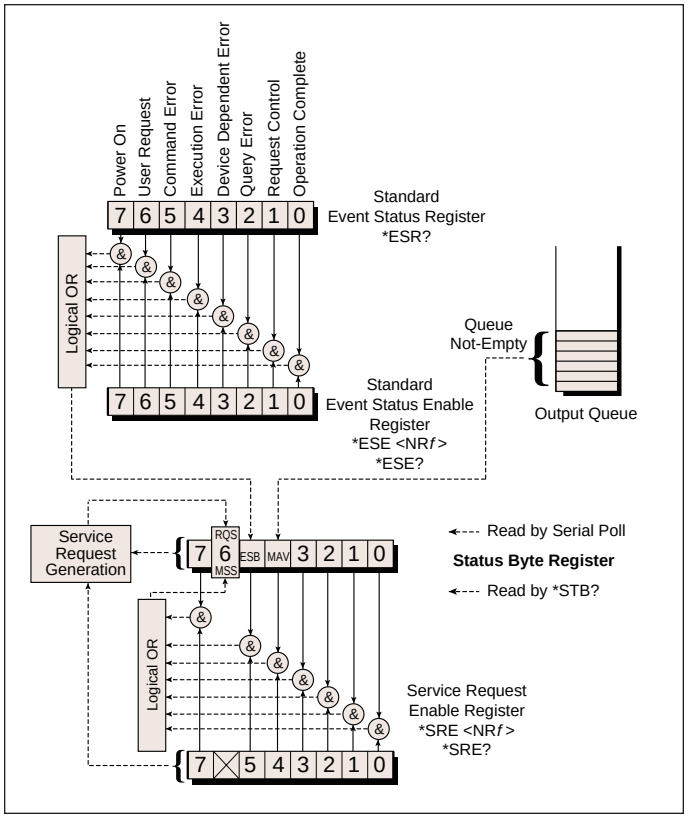
IFC (Interface Clear) — The Controller drives the IFC line to initialize the bus and become CIC.

REN (Remote Enable) — The Controller drives the REN line, which is used to place devices in remote or local program mode.

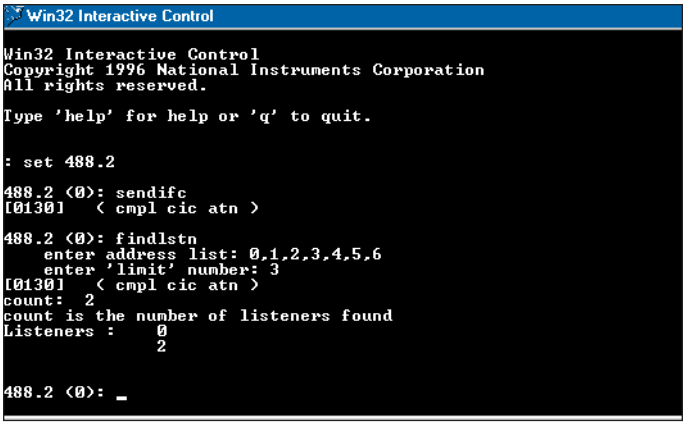
SRQ (Service Request) — Any device can drive the SRQ line to synchronously request service from the Controller.

EOI (End or Identify) — The EOI line has two purposes: The Talker uses the EOI line to mark the end of a message string, and the Controller uses the EOI line to tell devices to identify their response in a parallel poll.

The ANSI/IEEE Standard 488-1975, now called IEEE 488.1, greatly simplified the interconnection of programmable instrumentation by clearly defining



■ **Figure 3. IEEE 448.1 status reporting model.**



■ **Figure 4. Control screen for the IBIC control utility.**

mechanical, electrical, and hardware protocol specifications.

However, this standard does not address data formats, status reporting, message exchange protocol, common configuration commands, or device-specific commands. IEEE 488.2 enhanced and strengthened IEEE 488.1 by standardizing data formats, status reporting, error handling, Controller functionality and common commands to which all instruments must respond in a defined manner. The IEEE 488.2 standard focuses mainly on the software protocol issues and thus maintains compatibility with the hardware-oriented IEEE 488.1 standard. The IEEE 488.2 specifications for instruments can require major changes in the firmware and possibly the hardware.

However, IEEE 488.2 instruments are easier to program because they respond to common commands and queries in a well defined manner using standard message exchange protocols and data formats. IEEE 488.2 defines a minimum set of IEEE 488.1 interface capabilities that an instrument must have, like to be able to send and receive data, request service and respond to a device clear message. IEEE 488.2 standardized status reporting, and the Controller knows exactly how to obtain status information from every instrument in the system. This status reporting model builds upon the IEEE 488.1 status byte to provide more detailed status information. The status reporting model is shown in Figure 3.

Automating an Instrument (Overview, Visual Basic example, LabVIEW example)

Before reading or writing to an instrument, there is a common set of procedures to set up and communicate successfully to the instrument. The first step is to install the GPIB board. This is relatively easy, as it involves installing the right driver software (operating system dependent) and plugging in the GPIB board. If the GPIB board has an analyzer, you need to install the analyzer part of the GPIB software also. We used the PCI-GPIB card, which does not have analyzer capabilities. Once the card is installed properly inside the computer, you can run the diagnostics to test it. For plug and play devices, the diagnostics should pass. If the diagnostics fail, then there is usually a resource conflict with other devices in the computer. The next step is to open communication to the board. We used 488.2 compliant commands. In order to do this we used the ibic [2] software that comes with the National Instruments driver. From the prompt, type:

: ibfind gpib0 (this opens the board for board level commands)
gpib0: ibpad 0 (set the board's primary address to 0)
gpib0: ibrsc 1 (configure the board as system controller)
gpib0: ibsic (set interface clear to remove old errors and become controller-in-charge)
gpib0: ibsre 1 (place devices in remote programming mode)
gpib0: ibln PAD 0 (Find device at primary address PAD, PAD lies between 1 and 30. This must return true before proceeding. If it returns false, verify that the device is powered on, connected to the bus and at the address specified.)
gpib0: ibdev 0 PAD 0 13 1 0 (here PAD is the same integer as above)
ud0: (this new prompt is at the device level; ud is unit descriptor, increments as you open more devices)
ud0: ibwrt "*idn?" (*idn? is an identification command understood by many devices, but not all)

[0100] (status is returned after each call. Values less than 8000 indicate successful completion)

■ The Anritsu Wiltron 37xxx model Vector Network Analyzer is used in the example.

count: 5 (this is the number of bytes sent on the command, and indicates the device "handshook" that many times so it accepted the data)

ibrd 100 (reads from device, up to 100 characters)

You can also have ibic find all listeners (see Figure 4). However, this is a 488.2 command, so you must have a board (post-1990) which can support the commands:

: set 488.2 0 (activates 488.2 commands; 0 is board index)
488.2 (0): sendifc (clears the interface, makes gpib0 the Con-troller In Charge - CIC)
488.2 (0): findlstn
enter address list: 0,1,2,3,4,5,6,7 (possible addresses for devices, numbers between 0 and 30, separated by commas)
enter 'limit' number: 3 (maximum number of devices ibic will look for)
[0130] (cmpl cic atn) (transaction completed, board is controller in charge, attention asserted)
count: 2 (count is the number of listeners found)
listeners: **0** (controller's address)
2 (device's address)

This will help you determine if any devices are listening and what their addresses are. At this point, you can use ibfind or ibdev to go back to regular commands.

ud0: quit (this quits ibic)

Once you have established communication to the instrument, you can read and write data to it either from ibic or from another language interface. In the next section we will discuss using a Visual Basic and a LabVIEW example the commands we used and the different functionality of the VNA.

Example: Automating a VNA

The Vector Network Analyzer that we used was the Anritsu Wiltron 37xxx (see photo above). The GPIB address of the instrument was 6 and it had no secondary address. We used Windows 95 and used a National Instruments PCI-GPIB card. We followed the order specified in the previous section and the VNA responded to all the commands.

The next step is to go to the programming environment and start reading and writing to the instrument. The NI-488.2 driver will handle all the communication to the board.

If you are using Microsoft C/C++ Language Interface Files, they contain a documentation file (readme.txt) that has information about the C language interface; a 32-bit include file (decl-32.h) that contains NI-488 function and NI-488.2 routine prototypes and various predefined constants; and a 32-bit C language interface file (gpib-32.obj) that an application links with in

order to access the 32-bit DLL. If you are using Borland C/C++ Language Interface Files, the documentation file (readme.txt) contains information about the C language interface. In addition, a 32-bit include file (decl-32.h) contains NI-488 function and NI-488.2 routine prototypes and various predefined constants. A 32-bit C language interface file (borlandc_gpib-32.obj) links with an application in order to access the 32-bit DLL.

Microsoft Visual Basic Language Interface Files contain a file (readme.txt) with information about the Visual Basic language interface; a Visual Basic global module (niglobal.bas) with predefined constant declarations; and a Visual Basic source file (vbib-32.bas) with NI-488.2 routine and NI-488 function prototypes.

However, if you are using LabVIEW, you do not need to include any files, as the driver installs them in the appropriate folders.

Whatever the language interface, the commands to read and write are the same. The "Send" (ibwrt) command writes to the instrument and the "Receive" command reads the reply back from the VNA [3]. The prototype for the "Send" command is:

Format : C
void Send (int boardID, Addr4882_t address, void *buffer, long count, int eotmode)

Format : Visual Basic
CALL Send (boardID%, address%, buffer$, eotmode%)

Where,
boardID = The interface board number
address = Address of a device to which data is sent
buffer = The data bytes to be sent
count = Number of bytes to be sent
eotmode = The data termination mode: DABend, NULLend, or NLend

The "Send" command addresses the device described by address to listen and the interface board to talk. Then count bytes from buffer are sent to the device. The last byte is sent with the EOI line asserted if eotmode is DABend. The last byte is sent without the EOI line asserted if eotmode is NULLend. If eotmode is NLend, then a new line character ("\n") is sent with the EOI line asserted after the last byte of the buffer. The actual number of bytes transferred is returned in the global variable, ibcntl. The prototype for the "Receive" command is:

Format: C — **void Receive (int boardID, Addr4882_t address, void *buffer, long count, int termination)**

Format: Visual Basic — **CALL Receive (boardID%, address%, buffer$, termination%)**

Where,
boardID = The interface board number
address = Address of a device to receive data
count = Number of bytes to read
termination = Description of the data termination mode (STOPend or an EOS character)
buffer = Stores the received data bytes

The "Receive" command addresses the device described by address to talk and the interface board to listen. Count bytes are read and placed into the buffer. Data bytes are read until either count bytes have been read or the termination condition is detected. If the termination condition is STOPend, the read is stopped when a byte is received with the EOI line asserted. Otherwise, the read is stopped when an 8-bit EOS character is detected. The actual number of bytes transferred is returned in the global variable, ibcntl.

In LabVIEW you can use either a "GPIB Send" or a "GPIB Write" to pass the command to the instrument. The "GPIB Receive" command reads data back from the device. The parameters you would pass to the "write" and "receive" are the same for all language interfaces.

The following are the commands and the general format to use [4]. We will first discuss basic operation and then go on to get the S parameters. If you are only interested in the S parameters, you can skip the first section and go directly to the S parameter section [5].

- Reset instrument
  - Send reset command
    Send [6] (0,6, "*RST" , Nlend)
- Wait for operations to complete
  WaitForInstr ( )
  This function is called to change the timeout limit so that enough time is allowed for completing instrument operations. It uses the NI-488 function "Ibtmo( )".
  - Set GPIB timeout limit
    Ibtmo (instrument_handle, T1000s)
  - Send operation Complete query
    Send (0,6, "*OPC?", Nlend)
  - Wait for instrument to output ASCII character "1"
    Numbytes = 1
    Receive (0,6, buffer, Numbytes, STOPend)
  - Restore default timeout
    Ibtmo (instrument_handle, T10s)
- Setup display and sweep frequencies (active channel set to 2)
  Send (0,6, "CH2;DSP;MPH;SRT 40 MHZ;STP 20GHZ", NLend) [3]
- Setup markers
  Send (0,6, "MK1 40 MHZ;MK2 20 GHZ" , Nlend)
- Read and store current instrument setup
  - Request instrument setup string
    Send (0,6, "OFP", Nlend)
  - Read instrument setup string
    Receive (0,6, InstrSetup, MAXSIZE, STOPend)
  - Get number of bytes transferred
    SizeInstrSetup = IBCNT
  - Read sweep frequencies
    - Trigger and wait for full sweep then hold
      Send (0,6, "TRS;WFS;HLD", Nlend)

- Wait for operations to complete
  WaitForInstr ( )
- Read sweep frequencies(OFV)
  Use floating point (64 bit) Binary format (FMB), Most significant Byte first ordering (MSB)
  Send (0,6, "MSB;FMB;OFV", Nlend)
- Get number of bytes to read
  Numbytes = GetNumBytes (address, headerstring)

The GetNumBytes( ) function reads the 37xxx output buffer and returns the number of databytes to be transferred in the ensuing data string. It does this by reading and decoding the string data header. It will copy the header read out of the 37xxx into headerString so the calling program can use it in cases where the same data block will be sent back to the 37xxx. The format for the data that is returned by the receive is "#XYDT," where # is the first byte of the header, "X" is the value of the number of bytes that contain the location of the actual data, "Y" is the number of bytes that contain the data, "D" is the actual data and "T" is the termination character, if any.

- Read the first byte in the instrument output buffer. Buffer is a temporary array of characters of size 10
  NumBytes = 1
  Receive (0,6, buffer, NumBytes, STOPend)
- Check to be sure it is a "#" character, then copy it to headerstring
  If (buffer[0] ISNOTEQUALTO "#") then GPIBerr ("Invalid Data String") else Copy (buffer, headerstring)
- Read second header byte from the instrument output buffer and append it (concatenate) to headerstring.
  NumBytes = 1
  Receive (0,6, buffer, NumBytes, STOPend)
  Concatenate (buffer, headerstring)
- Save the buffer value as a number
  Numbytes = VALUEOF (buffer). This number is the next set of bytes to read. Those bytes when taken as a number will yield the number of actual data bytes contained in the binary string.
- Read the number of bytes indicated by NumBytes and append them to headerstring.
  Receive (0,6, buffer, NumBytes, STOPend)
  Concatenate (buffer, headerstring)
- Save the buffer value as a number
  Numbytes = VALUEOF(buffer). Now, NumBytes is the number of bytes of actual data requested, waiting in the output buffer of the 37xxx.
- Return the number of bytes to the calling program
  Return (NumBytes)
- Read frequencies
  FreqArray is a floating point, double precision array of up to 1601 elements.
  Receive (0,6, FreqArray, NumBytes, STOPend)
- Check for complete transfer
  If (NumBytes ISNOTEQUALTO IBCNT) then GPIBerr ("Could not read Frequency List correctly")
- Reset Instrument
  - Send reset command
    Send (0,6, "*RST" , Nlend)
  - Wait for operations to complete
    WaitForInstr ( )
- Select instrument Marker 1 active
  Send (0,6, "MR1", Nlend)
- Read measurement trace
  - Trigger and wait for full sweep then hold
    Send (0,6, "TRS;WFS;HLD", Nlend)
  - Wait for operations to complete
    WaitForInstr ( )
  - Request trace data
    The final graph type Values (OFD), use floating point binary format (FMB) and use most significant byte ordering.
    Send (0,6, "MSB;FMB;OFD" , Nlend)
  - Get number of bytes to read
    NumBytes = GetNumBytes
  - Read out the trace data values
    Receive (0,6, Tracedata, NumBytes, STOPend)
  - Check if all data was transferred
    If (NumBytes ISNOTEQUALTO IBCNT) then GPIBerr ("Could not receive data")
  - Calculate number of sweep points in data string
    POINTSIZE is 8 bytes for data transfers using FMB and 4 bytes with FMC.
    NumFreqs = NumBytes / POINTSIZE
- Get S_{11} data
  - Set instrument to active channel 2 and to set display mode to Linear magnitude and Phase display for reading S_{11}. The S_{11} command measures the forward reflection parameter S_{11}, on the active channel. Forward reflection is the value of the signal leaving port 1 vs the value of the signal reflected back into port 1.
    Send (0,6, "Ch2;S_{11};LPH" , Nlend)
  - Request frequencies
    Send (0,6, "MSB;FMB;OFV" , Nlend)
    Output current frequency sweep values (OFV), use floating point binary format (FMB) and use most significant byte ordering.
  - Read frequencies
    NumBytes = GetNumBytes( )
    Receive (0,6, Frequency, NumBytes, STOPend)
  - Request S_{11} information
    Send (0,6, "MSB;FMB;OFD" , Nlend)
  - Read S_{11} data. The S_{11} Data array is a complex array that has a real and imaginary part corresponding to magnitude and phase.
    Receive (0,6, S_{11}Data, NumBytes, STOPend)
- Get S_{21} data
  - Set instrument to active channel 4 and to set display mode to Linear magnitude and Phase display for reading S_{21}. The S_{21} command measures the forward transmission parameter S_{21}, on the active channel. Forward reflection is the value of the signal leaving port 1 vs. the value of the signal reflected back into port 1.
    Send (0,6, "Ch4;S_{21};LPH", Nlend)

- Request frequencies
  Send (0,6, "MSB;FMB;OFV" , Nlend)
  Output current frequency sweep values(OFV), use floating point binary format(FMB) and use most significant byte ordering.
- Read frequencies
  NumBytes = GetNumBytes( )
  Receive (0,6, Frequency, NumBytes, STOPend)
- Request S21 information
  Send (0,6, "MSB;FMB;OFD" , Nlend)
- Read S_{21} data. The S_{21}Data array is a complex array that has a real and imaginary part corresponding to magnitude and phase.
  Receive (0,6, S_{21}Data, NumBytes, STOPend)
- Put Instrument in local to allow front panel use
  EnableLocal (0,6)

If you are using LabVIEW the same commands apply. Simply use the GPIB write VI for sending data over and the GPIB receive. The data is received back in a string format which you can then convert to an array that can be used to calculate the data. To display the data on a Smith chart the command is "SMI," and to get a log magnitude and phase display the comand is "MPH" and so the mode of display can be changed programatically. Figure 5 shows the LabVIEWcode for extracting the S_{11} and S_{21} parameters.

Instrument drivers

Once you can control a VNA with direct commands, you must build an instrument driver, a set of software routines that control a programmable instrument. Each routine corresponds to a programmatic operation such as configuring, reading from, writing to and triggering the instrument. While instrument driver VIs can be run interactively, they do not contain continuous loops that read input settings and send instrument commands in response to real-time dynamic user input. Instead, they read the controls of the front panel, format and send command strings to the instrument, read the responses to instrument queries, and update front panel indicators *once* per VI execution.

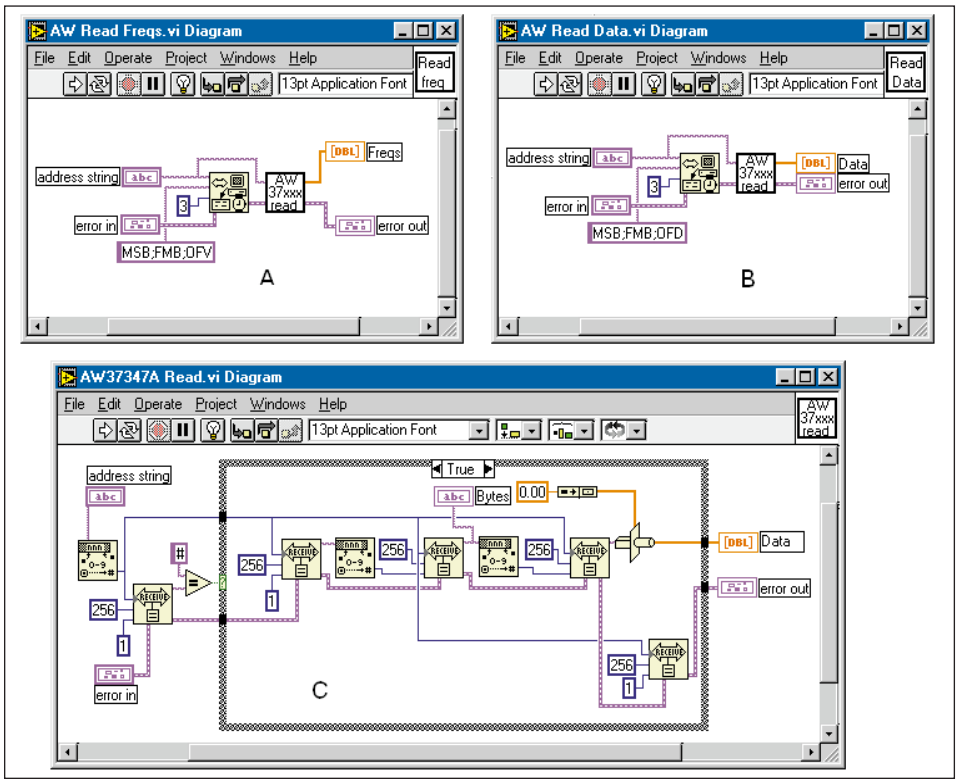
You can control several identical instruments at the same time using the same instrument driver if it is designed correctly. Instrument driver VIs, like all other LabVIEW VIs, are serially reusable. By calling an initialize VI several times with different addresses and then passing reference handles (or VISA sessions) between VIs, you can use the instrument driver VIs to control more than one instrument in an application [7].

Instrument driver architecture

Modern GPIB and VXIbus instruments are characterized by increasingly larger numbers of functions and modes. With this added complexity, it is necessary to provide a consistent design model that will aid both instrument driver developers and end users who develop instrument control applications. To define a standard for instrument driver software design and development, it is necessary to use conceptual models around which the design specifications are written. An external interface model will show how the instrument driver interfaces to other software components in the system. Similarly, an internal design model will define how an instrument driver software module is organized internally. An instrument driver consists of software modules, or VIs, that the user can call interactively as well as from a higher-level software application.

This general model contains the instrument driver *functional body*, which is the code for the instrument driver. The details for the functional body are explained in the internal design model. The *programmatic developer interface* is the mechanism for calling the driver from a high-level application program.

For example, a manufacturer's test system might make instrument driver calls to communicate with a multimeter or an oscilloscope. Therefore, the instrument driver sub-VIs would be used within a larger application. The *interactive developer interface* assists in the understanding of the function



■ **Figure 5. LabVIEW code for (a) reading frequencies, (b) reading the S-parameter data, and (c) low level VI to read all data.**

of each instrument driver VI. By running the front panels of the instrument driver sub-VIs, a developer can easily understand how to use the instrument driver in his application. The VISA (Virtual Instrument Software Architecture) *I/O interface* is the mechanism through which the driver communicates with the instrument hardware.

VISA is an established standard instrumentation interface for controlling GPIB, VXI, serial and other types of instruments from application software such as LabVIEW. The *subroutine interface* is the mechanism through which the driver can call support the VIs needed to accomplish a task. For example, cleanup and error messaging VIs are considered necessary support VIs.

Instrument driver internal design model

Each instrument driver has a number of VIs organized into a modular hierarchy containing not only high-level general-purpose application VIs, but also full-featured instrument driver component VIs.

The LabVIEW instrument driver internal design model, shown in Figure 2, defines the organization of the instrument driver *functional* body. This model is important to instrument driver developers because it is the foundation upon which the development guidelines are based. It is also important to end users because all LabVIEW instrument drivers are organized according to this model. Once you understand the model and how to use one instrument driver, you can use that knowledge for every LabVIEW instrument driver.

The functional body of a LabVIEW instrument driver consists of two main categories of VIs. The first category is a collection of *component VIs*, which are individual software modules that each control a specific area of instrument functionality. The second category is a collection of higher-level *application VIs*, which combine component VIs to perform basic test and measurement operations with the instrument. The internal design model of LabVIEW instrument drivers is built on a proven methodology. With this model, you have the necessary granularity to control instruments properly in your software applications. You can, for example, initialize all instruments once at the start, configure multiple instruments, and then trigger several instruments simultaneously. As another example, you can initialize and configure an instrument once, and then trigger and read from the instrument several times.

Instrument driver component VIs

The application VIs are built from a lower level set of instrument driver functions called component VIs. Unlike the application VI (which presents only a subset of the instrument features), component VIs are organized into a modular assortment containing all of the instrument configuration and measurement capabilities. The component VIs fit into six categories — *initialize, configuration, action/status, data, utility and close*.

Initialize — All LabVIEW instrument drivers should have an initialize VI. It is the first instrument driver VI called and it establishes communication with the instrument. Optionally, it can perform an instrument identification query and reset operations. It may also place the instrument either in its default power on state or in some other specific state.

Configuration VIs — The configuration VIs are a collection of software routines that configure the instrument to perform the desired operation. There are usually a number of configuration VIs depending on the complexity of the instrument. After these VIs are called, the instrument is ready to take measurements or stimulate a system.

Action/Status VIs — The action/status category contains two types of VIs. Action VIs cause the instrument to initiate or terminate test and measurement operations. These operations can include arming the triggering system or generating a stimulus. These VIs are different from the configuration VIs because they do not change the instrument settings; they simply order the instrument to carry out an action based on its current configuration. The status VIs obtain the current status of the instrument or the status of pending operations. Although the specific routines in this category and the actual operations they perform are at the discretion of the developer, they usually are created on a need basis as required by other functions.

Data VIs — The data VIs include VIs to transfer data to or from the instrument. Examples include VIs for reading a measured value or waveform from a measurement instrument, for downloading waveforms or digital patterns to a source instrument, and so on. The specific routines in this category depend on the instrument and are left up to the instrument driver developer.

Utility VIs — The utility VIs can perform a variety of operations that are auxiliary to the most often used instrument driver VIs. These VIs include the majority of the template instrument driver VIs (described below) such as reset, self-test, revision, and error query, and may include other custom routines such as calibration or storing/recalling instrument configurations.

Close VIs — All LabVIEW instrument drivers should include a close VI. The close VI terminates the software connection to the instrument and deallocates system resources.

Each of these categories, with the exception of initialize and close, contain several modular VIs. Most of the critical work in developing an instrument driver lies in the initial design and organization of the instrument driver component VIs. The specific routines in each category are further categorized as either template VIs or developer-specified VIs.

There are several advantages to using instrument drivers. Instrument drivers simplify instrument control and reduce test program development time by eliminating the need to learn the programming protocol for each instrument. Programming is much easier as they all follow similar guidelines. Because instrument driver VIs contain high level functions with intuitive front panels, end users can quickly test and verify the remote capabilities of their instrument without the knowledge of device-specific syntax. Also, the end user can easily cre-

ate instrument control applications and systems by programmatically linking instrument driver VIs in their block diagram. The LabVIEW Instrument Library contains instrument drivers for a variety of programmable instrumentation, including GPIB, VXI, RS-232/422, and CAMAC instruments.

Summary

Automating instruments in general helps with ease of use of these instruments as well as measurement efficiency. You can execute a series of steps without any user interface and you now have the power of the PC to manipulate the data that you have gathered. The hardware setup is simple and straightforward. Whatever your language interface, you can communicate to the instrument with standard syntax. When using these instruments on a day to day basis, you can use instrument drivers to easily control them. ■

Notes and references

1. In addition to the standard IEEE 488.1 three-wire handshake, National Instruments has developed the patented high-speed GPIB handshake protocol, called HS488, to increase the data transfer rate of a GPIB system up to 8 Mb/s. The HS488 high-speed mode is available on GPIB interfaces that use the TNT4882 chip.

2. The Win-32 interactive control utility is installed with the NI-488.2 driver and can be accessed from the start menu in Windows 95/98/NT.

3. Refer to the NI-488.2 Function reference manual for a complete list of commands and syntax.

4. Refer to the Series 37xxx Vector Network Analyzer Programming manual for a complete list of Instrument commands.

5. The code given here uses 488.2 compliant commands. Please consult your programming language documentation for the exact syntax of all other functions.

6. All GPIB functions are declared in bold.

7. For more details on multi-instance, please refer to the Advanced LabVIEW Instrument Driver Development Techniques application note, found on the National Instruments web page (http://www.natinst.com/support/).

Author Information

Jay Michael is an applications engineer at National Instruments, 11500 N. Mopac Expressway, Austin, Texas, 78759. He holds a masters degree in electrical engineering from the University of Arkansas and works with National Instruments' Test Executive products. He may be contacted by e-mail at jay.michael@natinst.com.