ICS ELECTRONICS
division of Systems West Inc.

IEEE 488 APPLICATION BULLETIN

GPIB 101 - A TUTORIAL ABOUT THE GPIB BUS

INTRODUCTION

The purpose of this application note is to provide guidance and understanding of the GPIB bus to new GPIB bus users or to someone who needs more information on using the GPIB bus's features. This application note is divided into short chapters. New GPIB bus users are encouraged to read all of the chapters. Experienced users may want to skip to the chapter that deals with their specific subject.

Application Note AB48-12, deals with bus extension problems and how to overcome the GPIB Bus limitations.

Please send all comments to appsengr@icselect.com. Let us know if there is a subject that needs more coverage or if there is something you feel we left out of this note.

CHAPTER 1 - HISTORY AND CONCEPT

The GPIB bus was invented by Hewlett-Packard Corporation in 1974 to simplify the interconnection of test instruments with computers. At that time, computers were bulky devices and did not have standard interface ports. Instruments had a connector with parallel BCD output lines that could be connected to a 10 to 20 column BCD printer. Data collection was mainly done by printing the current reading as a line on the printer. Remote control of an instrument was limited to a few input lines on a rear panel connector that selected a few functions or conversion ranges. A special computer interface had to be designed and built for each instrument that the engineer wanted to add to his test system. Building even the simplest automated test system was a several man-month project.

As conceived by HP, the new Hewlett-Packard Instrument Bus (HP-IB) would use a standard cable to interconnect multiple instruments to the computer. Each instrument would have its own interface electronics and a standard set of responses to commands. The system would be easily expandable so multi-instrument test systems could be put together by piggy backing cables from one instrument to another. There were restrictions on the number of instruments that a driver could drive (14) and the length of the bus cable (20 meters).

Hewlett-Packard proposed the concept to US and International standards bodies in 1974. It was adopted by the IEC committee in Europe in 1975. In the United States, other instrument companies objected to the HP-IB name and so a new name, the General Purpose Instrument Bus (GPIB) was created. The GPIB bus was formally adopted as the IEEE-STD 488 in 1978.

The IEEE-488 concept of Controllers and Devices is shown in Figure 1. Controllers have the ability to send commands, to talk data onto the bus and to listen to data from devices. Devices can have talk and listen capability. Control can be passed from the active controller (Controller-in-charge) to any device with controller capability. One controller in the system is defined as the System Controller and it is the initial Controller-in-Charge (CIC).

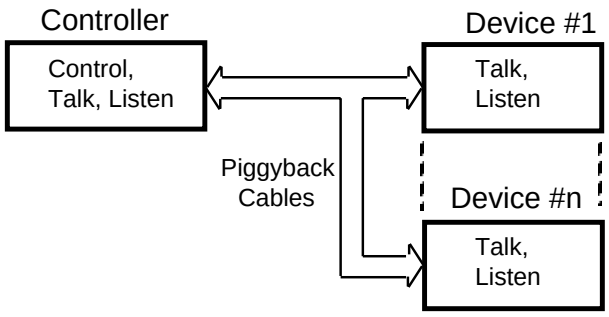

Figure 1 IEEE 488 Bus Concept

Devices are addressable as talkers and listeners and have to have a way to set their address. Each device has a primary address between 0 and 30. Address 31 is the Unlisten or Untalk address. Devices can also have secondary addresses that can be used to address device sub-functions or channels. An example is ICS's 4896 GPIB-to-Quad Serial Interface which uses secondary addresses to address each channel. Although there are 31 primary addresses, IEEE 488 drivers can only drive 14 physical devices.

Revised 10-18-03

Some devices can be set to talk only or to listen only. This lets two devices communicate without the need for a controller in the system. An example is a DVM that outputs readings and a printer that prints the data.

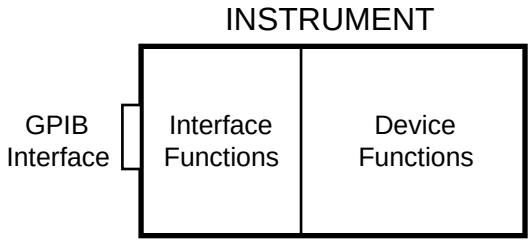

Figure 2 IEEE-488 Instrument

The IEEE-488 Standard defined an instrument with interface and device partitions as shown in Figure 2. Interface messages and addresses are sent from the controller-in-charge to the device's interface function. Instrument particular commands such as range, mode etc., are data messages that are passed through the Interface to the device.

Physical Bus Structure

Physically the GPIB bus is composed of 16 low-true signal lines. Eight of the lines are bidirectional data lines, **DIO1-8**. Three of the lines are handshake lines, **NRFD**, **NDAC** and **DAV**, that transfer data from the talker to all devices who are addressed to listen. The talker drives the DAV line, the listeners drive the NDAC and NRFD lines. The remaining five lines are used to control the bus's operation.

ATN (attention) is set true by the controller-in-charge while it is sending interface messages or device addresses. ATN is false when the bus is transmitting data.

EOI (end or identify) can be asserted to mark the last character of a message or asserted with the ATN signal to conduct a parallel poll.

IFC (interface clear) is sent by the system controller to unaddress all devices and places the interface function in a known quiescent state.

REN (remote enable) is sent by the system controller and used with other interface messages or device addresses to select either local or remote control of each device.

SRQ (service request) is sent by any device on the bus that wants service.

Interface Messages

Table 1 lists the GPIB Interface Messages and Addresses with their common mnemonics. MLA, LAD and UNL are listen addresses with hex values of 20 to 3F. MTA, TAD and UNT are talk addresses with hex values of 40-5F. A device normally responds to both talk and listen addresses with the same value. i.e. LAD 4 and TAD 4. Secondary addresses have hex values of 60-7F.

Devices are designed with different IEEE 488 capabilities so not all devices respond to all of the interface messages. Universal messages are recognized by all devices regardless of their address state. Addressed commands are only recognized by devices that are active listeners.

Table 1 488.1 Interface Messages and Addresses

Command	Function
Address Commands	
MLA	My listen address (controller to self)
MTA	My talk address (controller to self)
LAD	Device listen address (0-30)
TAD	Device talk address (0-30)
SAD	Secondary Device address (device optional address of 0-31)
UNL	Unlisten (LAD 31)
UNT	Listen (TAD 31)
Universal Commands (to all devices)	
LLO	Local Lockout
DCL	Device Clear
PPU	Parallel Poll Unconfigure
SPE	Serial Poll Enable
SPD	Serial Poll Disable
Addressed Commands (to addressed listeners only)	
SDC	Selected Device Clear
GTL	Go to Local
GET	Device Trigger
PPC	Parallel Poll Configure
TCT	Take Control

The Standard also defined a Status Byte in the instrument that could be read with a Serial Poll to determine the device's status. Bit 6 of the Status Byte was defined as the Service Request bit that could be set when other bits in the Status Byte are set. The other bits were user defined. The Service Request pulls the SRQ line low to interrupt the controller. The Service Request is reset when the device is Serial Polled or when the service request cause is satisfied.

488.2 STANDARD

The GPIB concept expressed in IEEE-STD 488 made it easy to physically interconnect instruments but it did not make it easy for a programmer to talk to each instrument. Some companies terminated their instrument responses with a carriage return, others used a carriage return-linefeed sequence, or just a linefeed. Number systems, command names and coding depended upon the instrument manufacturer. In an attempt to standardize the instrument formats, Tektronix proposed a set of standard formats in 1985. This was the basis for the IEEE-STD 488.2 standard that was adopted in 1987. At the same time, the original IEEE-488 Standard was renumbered to 488.1.

The new IEEE-488.2 Standard established standard instrument message formats, a set of common commands, a standard Status Reporting Structure and controller protocols that would unify the control of instruments made by hundreds of manufacturers.

The standard instrument message format terminates a message with a linefeed and or by asserting EOI on the last character. Multiple commands in the same message are separated by semicolons. Fixed point data became the default format for numeric responses.

The common command set defined a subset of ten commands that each IEEE-488.2 compatible instrument must respond to plus additional optional commands for instruments with expanded capabilities. The required common commands simplified instrument programming by giving the programmer a minimal set of commands that he can count on being recognized by each 488.2 instrument. Table 2 lists the 488.2 Common Commands and their functions. Probably the most familiar Common Command is the *IDN? query. This is a good first command to use with an instrument as its response shows what the instrument is and demonstrates that you have communication with the instrument. The most of the remaining commands are used with the Status Reporting Structure.

The IEEE-488.2 Standard Status Reporting Structure is shown in Figure 3. The new Status Reporting Structure expanded on the 488.1 Status Byte by adding a Standard Event Status Register (ESR Register) and an Output Queue. Enable registers and summation logic was added to the Status Registers so that a user could enable selected bits in any status register.

The ESR Register reports standardized device status and command errors. Bit 6 in the ESR Register is not used and can be assigned for any use by the device designer. The Standard Event Status Enable Register is used to select which event bits are summarized into the Status Byte. When an enabled bit in the Event Status Register becomes true, it is ORed into the summary output which sets the ESB bit (bit 5) in the Status Byte Register. Bits in the ESR Register stay set until the register is read by the *ESR? query or cleared by the *CLS command.

The Output Queue contains responses from the 488.2 queries. Its status is reported in the MAV bit (bit 4) of the Status Byte. Typically this bit is not enabled because the user normally follows a query by reading the response.

The 488.2 Status Byte contains the ESB and MAV bits plus five user definable bits. Bit 6 is still the RQS bit but it now has a dual personality. When the Status Byte is read by a Serial Poll, the RQS bit is reset. When the Status Byte is read by the *STB? query, the MSS bit is left unchanged. Service Request generation is a two step process. When an enabled bit in the ESR Register is set, the summary output sets the ESB bit in the Status Byte Register. If the ESB bit is enabled, then the RQS bit is set and a SRQ is generated. Reading or clearing the ESR Register, drops the summary output which in turn, resets the ESB bit in the Status Byte. If no other enabled bits in the Status Byte are true, bit 6 and the SRQ line will be reset

Saving the Device Configuration

488.2 and SCPI compliant devices accept commands whose values are saved in an internal nonvolatile memory. The 488.2 *SAV 0 command is used to save the values. The device may also save its current output settings along with the configuration values so be sure that all outputs are in the desired state before sending the device the *SAV 0 command.

Table 2 488.2 Common Commands

Command	Function
Required common commands are:	
***CLS**	Clear Status Command
***ESE**	Standard Event Status Enable Command
***ESE?**	Standard Event Status Enable Query
***ESR?**	Standard Event Status Register Query (0-255)
***IDN?**	Identification Query (Company, model, serial number and revision)
***OPC**	Operation Complete Command
***OPC?**	Operation Complete Query
***RST**	Reset Command
***SRE**	Service Request Enable Command
***SRE?**	Service Request Enable Query (0-255)
***STB?**	Status Byte Query Z (0-255)
***TST?**	Self-Test Query
***WAI**	Wait-to-Continue Command
Devices that support parallel polls must support the following three commands:	
***IST?**	Individual Status Query?
***PRE**	Parallel Poll Register Enable Command
***PRE?**	Parallel Poll Register Enable Query
Devices that support Device Trigger must support the following commands:	
***TRG**	Trigger Command
Controllers must support the following command:	
***PCB**	Pass Control Back Command
Devices that save and restore settings support the following commands:	
***RCL**	Recall configuration
***SAV**	Save configuration
Devices that save and restore enable register settings support the following commands:	
***PSC**	Saves enable register values and enables/ disables recall
***PSC?**	PSC value query

Saving the Enable Register Settings

The enable register settings cannot be saved with the *SAV 0 command. The 488.2 Standard defined a PSC flag which enables clearing the ESE and SRE registers at power turn-on. The enable registers are restored to a 0 value at power turn-on when the PSC flag is set on. The *PSC 0 command disables the PSC flag and saves the enable register values. The following example saves the current SRE and ESE settings. e.g.

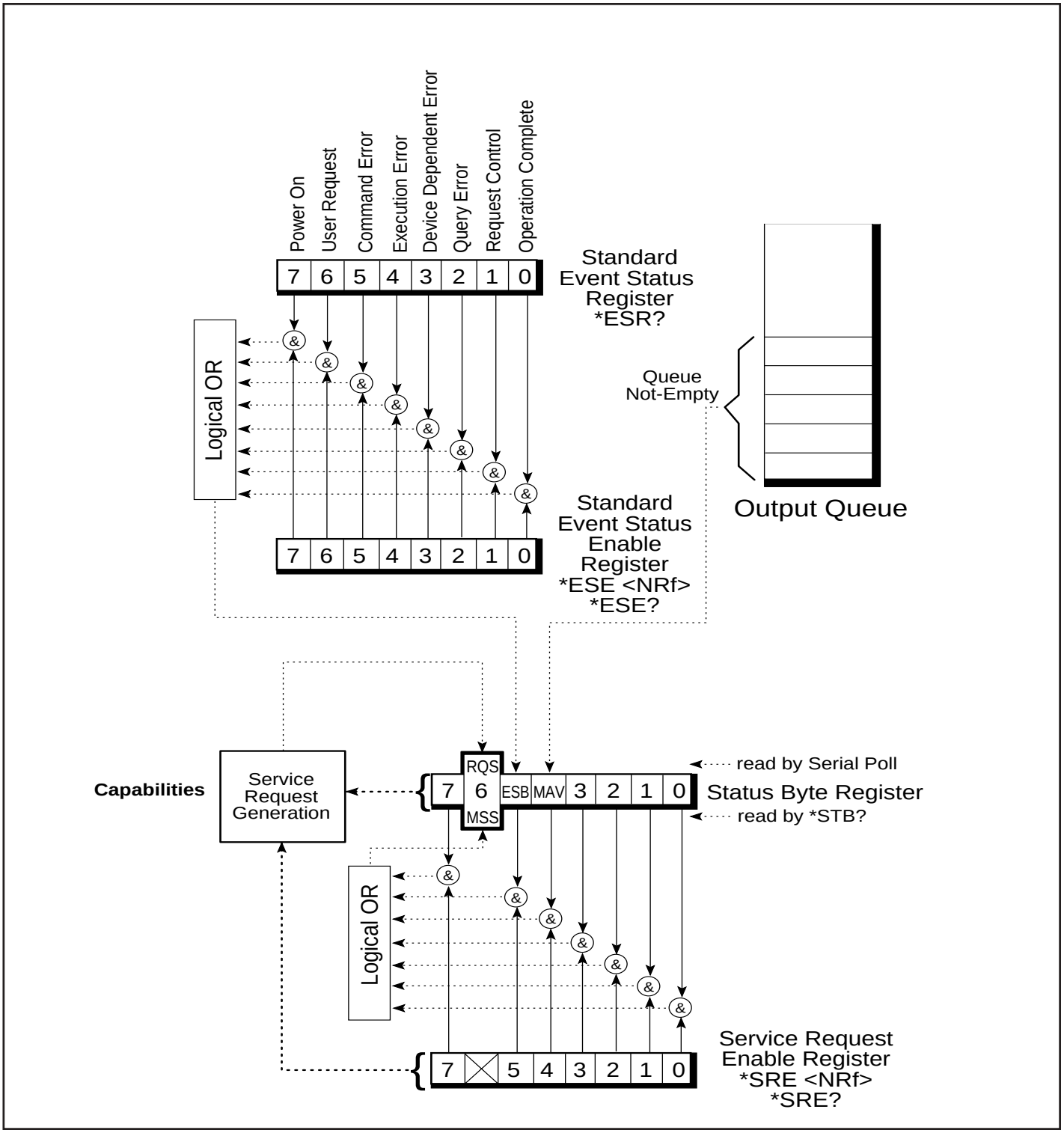

Figure 3 488.2 Status Reporting Structure

ESE 192; SRE 32; *PSC 0'saves ESE and SRE settings as the power on settings.

Note that a later *PSC 1 command sets the PSC flag which will cause the registers to be cleared at the next power turn-on and revert to their default values.

488.2 differences from 488.1

The 488.2 Standard downgraded the use of the Device Clear command so that it does not reset a device's outputs and internal memory as might be expected for a 488.1 device. Instead, check the device's manual and use an *RST or an *RCL 0 command to reset a 488.2 device and restore its power turn-on condition.

Common Controller Protocols

The 488.2 Standard defined several protocols that a 488.2 compliant GPIB controller would execute. The protocols are essentially subroutines that operate on all of the GPIB devices connected to the system. These protocols operate in systems that contain 488.2 compliant devices. Table 3 lists the 488.2 Common Controller Protocols. The Reset protocol and the AllSpoll protocol are mandatory for all 488.2 GPIB Controllers. FindLstn is probably the most used protocol as it finds and lists all of the devices on the bus. FindLstn is typically used at the start of a program to verify that the needed devices are connected to the system.

Table 3 488.2 Common Controller Protocols

Keyword	Function
ALLSPOLL	Serial polls all devices on the bus
FINDLSTN	Finds and lists all of the devices on the bus
FINDRQS	Finds the first device asserting SRQ
PASSCTL	Passes control of the bus
REQUESTCLT	Requests control of the bus
RESET	Resets all bus devices
SETADD	Sets a devices's GPIB bus address
TESTSYS	Self-tests the system

SCPI COMMANDS

The 488.2 Standard had made it easier to communicate with a GPIB instrument but each instrument still had a unique command set Even in a family of instruments from the same manufacturer, different instruments often had different command sets, reflecting the ideas of the instrument designer. The US Air Force recognized this problem in the early 1980s and initiated Project Mate to try to overcome this problem. The Mate Project concept was to create Translation Module Adapters (TMAs) to convert instrument unique commands into the Air Force's CIIL language. The TMAs could be external hardware adapters or internal firmware converters. The Air Force's CIIL language was a subset of Atlas. Its drawback was that it did not address the instrument model and it had very clumsy construction. The other part of the problem was that CIIL language instruments, when developed, did not sell well enough to justify the development costs.

Hewlett-Packard worked on the problem and in 1990, proposed a Test Measurement Language (TML) that was based on an instrument model. It was a tree-branch type of language that allowed the same commands to be used for instruments from different manufacturers. TML included major command branches that could control virtually all of an instrument's subsystems. It was an open structure so that other manufacturers could add commands they felt necessary. HP offered to licensed TML to any instrument manufacturer for a small fee and a pledge to obey the TML specification. Almost immediately, Tektronix and some other companies objected to HP controlling the TML specification. HP promptly offered it to a consortium which rechristened it as Standard Commands for Programmable Instruments (SCPI).

SCPI commands use common command words (keywords) and programming syntax to give all instruments a common "look and feel". Control of any instrument capability that is described in SCPI should be implemented exactly as specified. Guidelines are included for adding new commands in the future without causing programming problems.

The obvious benefit of SCPI for the ATE programmer is in reducing of the learning time for programming multiple SCPI instruments since they all use a common command language and syntax. A second benefit of SCPI is that its English like structure and words are self documenting, eliminating the needs for comments explaining cryptic instrument commands. A third benefit is the interchangeability of newer SCPI instruments for older models or for another manufacturer's instrument with the same capabilities, and a reduction in programming maintenance when devices wear out and need replacement.

SCPI Command Structure and Examples

SCPI commands are based on a hierarchical structure that eliminates the need for most multi-word mnemonics. Each key word in the command steps the device parser out along the decision branch - similar to a squirrel hopping from the tree trunk out on the branches to the leaves. Subsequent keywords are considered to be at the same branch level until a new complete command is sent to the device. SCPI commands may be abbreviated as shown by the capital letters in Figure 4 or the whole key word may be used when entering a command. Figure 4 shows some single SCPI commands for setting up and querying a serial interface.

```
SYSTem:COMMunicate:SERial:BAUD 9600 <nl>
                              'Sets the baud rate to 9600 baud

SYST:COMM:SER:BAUD? <nl> 'Queries the baud rate

SYST:COMM:SER:BITS 8 <nl>   'Sets 8 data bits
```

Figure 4 SCPI Command Examples

Adventurous users may concatenate multiple SCPI commands together on the same line using semi colons as command separators. The first command is always referenced to the root node. Subsequent commands are referenced to the same tree level as the previous command. Starting the subsequent command with a colon puts it back at the root node. IEEE 488.2 common commands and queries can be freely mixed with SCPI messages in the same program message without affecting the above rules. Figure 5 shows some compound command examples.

```
SYST:COMM:SER:BAUD 9600; BAUD? <nl>

SYST:COMM:SER:BAUD 9600; :SYST:COMM:SER:
    BITS 8 <nl>

SYST:COMM:SER:BAUD 9600; BAUD?; BITS 8;
BITS?; PACE XON; PACE?<nl>
```

Figure 5 Compound Command Examples

A typical response to the last command example in Figure 5 would be: **9600;8; XON<nl>**

The response includes three items because the command contains three queries. The first item is **9600** which is the baud rate, the second item is **8** (bits/word) which is the current setting. The third item **XON** means that XON is active. It is always a good idea to check the devices' error light or read its ESR register with the ***ESR?** query when first using a compound command with a device to be sure that it was accepted by the device's parser.

SCPI Variables and Channel Lists

SCPI variables are separated by a space from the last keyword in the SCPI command. The variables can be numeric values, boolean values or ASCII strings. Numeric values are typically decimal numbers unless otherwise stated. When setting or querying register values, the decimal variable represents the sum of the binary bit weights for the bits with a logic '1' value. e.g. a decimal value of 23 represents 16 + 4 + 2 + 1 or 0001 0111 in binary. Boolean values can be either 0 or 1 or else OFF or ON. ASCII strings can be any legal ASCII character between 0 and 255 decimal except for 10 which is the Linefeed character.

Channel lists are used as a way of listing multiple values. Channel lists are enclosed in parenthesis and start with the ASCII '@' character. The values are separated with commas. The length of the channel list is determined by the device. A range of values can be indicated by the two end values separated by a colon. There is a space between the '@' and the first value. e.g.

(@1,2,3,4)	'lists sequential values
(@ 1:4)	'shows a range of sequential values
(@ 1,5,7,3, 4)	'lists random values

Figure 6 Channel List Examples

SCPI Error Reporting

SCPI provides a means of reporting errors by responses to the **SYST:ERR?** query. If the SCPI error queue is empty, the unit responds with 0, "No error" message. The error queue is cleared at power turn-on, by a ***CLS** command or by reading all current error messages. The error messages and numbers are defined by the SCPI specification and are the same for all SCPI devices.

CHAPTER 2 - GETTING STARTED

Now that you have a background on the GPIB bus you probably want to know how to use the GPIB bus to get things done. This Chapter deals with some general suggestions for putting the system together.

Bus Controllers

Most GPIB Bus controllers now are PCs with an add-on GPIB Controllers. These add-on controllers take the following forms:

1. ISA or PCI Cards installed in PCs
2. PCMCIA Cards in Portable PCs
2. Serial Port to GPIB Controllers
3. USB to GPIB Controllers

Cards installed in PCs have the benefit of being the lowest cost form of the GPIB Controller. This category includes ISA, AT, and PCI bus cards. PCI cards are the most popular as there is no longer any demand for ISA and AT bus cards. GPIB Controller cards are also available for the PC/104 bus. Cards can be obtained from Hewlett-Packard, ICS Electronics, IOtech, Measurement Computing and National Instruments.

PCMCIA cards are available for portable or laptop computers. There popularity is waning as the cable connectors are too delicate for rugged applications. The USB to GPIB Controllers are now more popular than the PCMCIA cards. PCMCIA Card GPIB Controllers can still be obtained from Agilent, ICS Electronics, IOtech, Measurement Computing and National Instruments.

Serial-to GPIB Controllers can be connected to a Computer's COM port or run at remote location by being connected to a modem and phone line. Some Serial-to GPIB Controllers have RS-422/RS-485 interfaces and can be run at the end of a very long serial cable. These Serial-to GPIB Controllers are convenient for adding a GPIB controller to a older portable computer that does not have a PCMCIA slot. Serial-to-GPIB Controllers can be obtained from ICS Electronics, IOtech, and National Instruments.

ICS and National Instruments make GPIB-to-Serial Converters that can be turned around and used in a serial-to-GPIB (S) mode to control a single device. This converter is not recommend for general use since it has only limited control over only one GPIB device. They are okay for one-on-one use or in an embedded applications.

USB to GPIB Controllers can be connected to the USB connector on the newer Desktop PCs and portable PCs. Requirements are that the computer must be running Windows 98 or Windows 95B (an OEM version of Windows 95). USB Controllers are available from ICS Electronics, Agilent and National Instruments in 1999.

Whichever GPIB controller you have selected, now is the time to install it. Install and test it in accordance with the manufacturer's instructions.

GPIB Bus Cables

Standard GPIB cables can be obtained from a number of sources including your GPIB Controller card manufacturer. They are available with the standard piggyback connectors at both ends or with a straight-in cable connector on one end of the cable. Use good quality multi-shielded cables to avoid EMI/RFI problems. If in doubt, ask your GPIB cable provider if the cables have passed a CE test. If so, they should be able to provide you with a CE certificate.

Total GPIB cable length in a system should not exceed 20 meters. For maximum data transfer rates, cable length should not exceed 2 meters between devices.

GPIB cables are often shipped with a 'brightener' on the connector contracts. This 'brightener' is a waxy organic substance to keep the contacts bright. If you start getting bad data, clean the GPIB connector contacts. Use a mild soap solution (a couple of drops of a dish detergent in a cup of water) to wash off the brightener. Clean the contacts with alcohol and blow dry the connector.

Device Addresses

GPIB devices can be assigned any primary address between 0 and 30. Assign a different address to each GPIB device. Avoid using addresses 0 and 21 as these may be used by the GPIB controller. GPIB controllers have their own GPIB address. (The GPIB address is software set and is not the GPIB controller card's internal PC bus address). National Instruments' controllers typically use GPIB address 0. HP and ICS Electronics controllers typically use GPIB address 21. Also don't use address 31 as a GPIB device address as 31 is the Unlisten and Untalk address.

GPIB Devices typically use a dip switch with five rockers to set the GPIB address. The rocker bit weights are 16, 8, 4, 2, and 1. Other rockers may set talk-only or listen-only modes and should be left off for use in a system with a GPIB Bus Controller. Always reset the instrument or power it off and back on after changing its address setting.

Some GPIB devices use front panel controls to set their GPIB addresses. These devices save the GPIB address in a nonvolatile memory. Follow the manufacturer's instructions when changing their GPIB address setting.

Some newer GPIB devices like ICS's Minibox interfaces use SCPI commands to change and set their GPIB bus address. Use the 'SYST:COMM:GPIB:ADDR aa' command where aa is the new GPIB address to change the address setting. The address change is immediate. Next, send the device the '*SAV 0' command at its **new** GPIB address to save the new address.

Interactive Keyboard Control Programs

Keyboard control programs are programs that let you interactively control and query a GPIB device by entering device related commands on the PC keyboard. The better programs do most of the work for you so you do not have to know the GPIB command syntax to use them. Some GPIB Controllers are supplied with these kind of programs If you have one, use it to check out your GPIB controller and devices before writing your program.

ICS's GPIBkybd program is a graphical Windows program that let the user control GPIB devices by simply entering a device specific message in a text box and by clicking on buttons to send the message and/or perform a simple command like Serial Poll or Device Clear. ICS's GPIBKybd program runs the 488.2 FindLstn protocol to find your GPIB device(s) when the program is started. The found devices are listed in the Response box and the program sets the device address to the lowest found device address. ICS's GPIBKybd program interfaces with the GPIB32.DLL so it can operate GPIB controllers cards from ICS Electronics, Measurement Computing, National Instruments and any other manufacturer who supplies an equivalent GPIB-32.DLL. You can download a free copy of ICS's GPIBKybd program from ICS's website at http://www.icselect.com/prgupdates.html.

Older control programs like National Instruments' ibic program are low level, DOS command line programs that use NI's older ib commands to control GPIB devices. National Instruments' ibic program requires you enter the controller card's 'ib' type command to communicate with the device.

Using a Keyboard Control Program

The best way to start with a Keyboard Controller program is to start with a known good device. Its best if it is an IEEE-488.2 compatible device.

1. Turn on the device and start the program
2. Set the program to the same address as the test device. ICS's GPIBKybd program should have found the device when it was started. For NI's ibic, do an ibfind to get the device handle.
3. Send the device an IFC to clear the GPIB interface by clicking the IFC button. For NI's 'ibic', do a ibsre(0,1) and a SendIFC(0).
4. Send the device the *idn? query and read back the device response. If there is no response, the device is probably not 488.2 compatible.
5. Perform a Serial Poll to see if the device can respond the GPIB controller. Repeat the serial poll one more time if the first response was not 0.
6. Once you have proved that you have communication with the GPIB device, you can send it device specific commands and read back its responses. The device specific commands are found in the programming section of the device's instruction manual.

CHAPTER 3 - WRITING GPIB PROGRAMS

GPIB programs are simple programs with three major sections: Initialization, main body and the closing. Most programs use just 6 to 8 commands so it is not necessary to learn all of the GPIB commands to develop a good GPIB program. Program complexity increases with the number of devices being controlled and number of tests. Programs can be written in C/C++ or in Visual Basic. Test programs can also be developed with graphical languages such as National Instrument's LabVIEW, Agilent's VEE or Measurement Computing's SoftWire.

This chapter provides directions for initializing the GPIB Controller and bus and for sending data to or reading data back from a GPIB device. Program examples are shown in BASIC syntax with ICS Electronics and with National Instruments' style commands. (NI commands include their original 488 command set and the newer 488.2 command set.) The directions given can be applied to other manufacturer's command sets.

GPIB Command Concept

GPIB commands in a program are like the layers of an onion. The inner layer is the device specific command that you want the device to have. i.e.

```
*RST, *IDN? or SYST:COMM:SERIAL:BAUD 9600
```

The next layer is the command required by your GPIB controller card to send or receive data or carry out some action on the GPIB bus. Examples are:

```
ieOutput(DevAddr%, "*RST")
ieEnter(DevAddr%, Rdg$)
or ieTrigger(DevAddr%)
```

The third layer is the calling convention that your programming language or programming style dictates. Some commands can be called with the standard CALL command. Other commands or languages equate an error variable to a return value that indicates if the command was successfully executed. Some examples are:

```
CALL ieOutput(DevAddr%, OutputStr$, Len)
ioerr% = ieOutput(DevAddr%, OutputStr$, Len)
```

Error Processing

Most GPIB command libraries have a way of determining whether the command was successfully executed by the GPIB controller. This does not mean that the device did what you wanted, just that the GPIB controller got the command to the device.

The error variable is set when the command is executed. The variable can be set by equating it to the command because the command returns the error variable. In other cases, the error is a separate variable that can be checked. It is a good plan to test the error variable after every command to be sure there were no problems with the command or with the device. This is done by adding a conditional test to the program after each instruction.

```
CALL ieOutput(DevAddr%, OutputStr$, Len)
ioerr% = ieOutput(DevAddr%, OutputStr$, Len)
Call ProcessError(ioerr%)
```

In the above example, the subroutine ProcessError tests the variable. If ioerr% is not zero, ProcessError will displays the appropriate error message in a box to the user. Examples of the Process Error routine are found in ICS's example Visual Basic programs.

```
Call = Send(Bd, Addr, OutString$, EOTMode)
If (ibsta AND EERR) then
    Call ReportError(Addr, " Did not respond")
End If
```

In this example, the test was done in the program. If the error variable , ibsta, was true, then ReportError was called to display the error.

GPIB Controller Initialization

The GPIB Controller Card is initialized to be sure that it is the System Controller and Controller-in-charge of the bus. The bus is initialized to be sure that all of the devices are in a non-addressed state after their power turn-on. This is done by having the GPIB Controller issue an Interface Clear command (IFC pulse) and assert the REN line. It is also a good idea to check or set the bus timeout. The bus timeout is the amount of time that the program will wait for a device to respond to a command before proceeding.

In the 488-PC2 Command Set, this is done with the following commands:

```
ioerr% =ieInit(IOPort, MyAddr, Setting) 'initializes the
                                         Driver
ioerr% =ieAbort                 'sends IFC, sets REN on
ioerr% = ieTimeOut (Time)       'sets bus timeout
```

The 488-PC2 Command Set returns an error status value in the ioerr% variable when it is finished. If the value is zero, the command was successfully executed. A nonzero value means the command was not executed correctly and that the device probably was not there and/or did not receive the Output String. When error variables are used, each command should be followed with a step that calls a routine to check the error variable. i.e.

```
ioerr% =ieAbort                 'sends IFC, sets REN on
Call ProcessError (ioerr%)      'checks the error variable
```

In the NI 488.2 Command Set, the initialization is done with the following commands:

```
Call SendIFC(Bd%)               'sends IFC
Call ibsre(Bd%, 1)              'sets REN on
Call ibtmo(Bd%, T3s)            'sets Timeout to 3 seconds
```

The NI 488 command, ibsre, is used to set REN because there is not an equivalent NI 488.2 command. The NI 488 command, ibtmo, is used to set the timeout because there is not equivalent NI 488.2 command. T3s is a predefined constant for 3 seconds. If timeout is set to 0 (or TNONE), the GPIB bus (and your program) will be

held as long as it takes for a device to complete its instruction. The 0 setting is not recommended except when debugging hardware.

NI 488 commands can be included in an NI 488.2 program when there is not an equivalent NI 488.2 function. The NI 488.2 Command Set returns errors in the command status in iberr and sets ibsta. ibsta should be checked after every command to be sure the command was executed correctly.

Sending Data to a Device

Data or device commands are normally sent to a device as strings of ASCII characters. Typical device commands are DVM setup commands, baud rate commands to a GPIB-to-Serial converter, or digital data to a Parallel interface.

In the 488-PC2 Command Set, ASCII data is sent by first specifying the string and then calling the ieOutput command:

```
Outstring$ = "Command to be sent"
L = Len(Outstring$)
ioerr% = ieOutput(DevAddr, OutString$, L)
```

In the NI 488.2 Command Set, ASCII data is sent by specifying the Output String and then calling the Send command.

```
Outstring$ = "Command to be sent"
Call = Send(Bd, Addr, OutString$, EOTMode)
```

You can also embed the command string in the call line. e.g.

```
Call = Send(Bd, Addr, "Command to be sent", EOTMode)
```

EOTMode is a flag that tells the Send command how to terminate the command string. All IEEE 488 command strings need to be terminated with a linefeed character or by asserting the EOI line on the last character. If the output data is binary data, terminate the output by only asserting EOI on the last character. NLend is a predefined constant that sends a linefeed with EOI asserted after the last data character. The above command then becomes:

```
Call = Send(Bd, Addr, OutString$, NLend)
```

Reading Data from a Device

ASCII data strings are read from a device by first specifying an empty string and then reading the data into the string. Data is read until a terminator is found or the defined Input string is full. Typical terminators are linefeed or EOI asserted on the last character. An example in the 488-PC2 Command Set is:

```
Instring$ = String$(Lin, 32)      'fills the string with spaces
ioerr% = ieEnter(DevAddr, Instring$, Lin)
```

where Lin is the length of the input buffer. The 488-PC2 Command Set uses the ieEOL command to set the input terminator. The ieEOL command defaults to LF or EOI sensed.

In the NI 488.2 Command Set the input example is:

```
Instring$ = String$(Lin, 32)      'fills the string with spaces
ioerr% =Receive(Bd, Addr,  Instring$, Term)
```

Term or Termination is the flag used to signal the end of the data. Term can be set to any ASCII character between 0 and FF HEX and the Receive process will stop when that character is detected. If Term is set to the predefined STOPend constant, the Receive process stops when EOI is detected.

Clearing a Device

Some devices have buffers that accumulate unwanted data and it occasionally becomes necessary to clear out the old data or return a device to a known condition. This is done by sending the device the Device Clear Command. In the 488-PC2 Command Set this is done with:

```
ioerr% = ieDevClr(DevAddr)
```

In the NI 488.2 Command Set, this is done with :

```
Call DevClear(Bd, Addr)
```

Reading the Device Status

Some times it is desirable to read the device's Status Register to see if it has data, has a problem or has completed some task. Devices report their status (Status Register contents) in response to Serial polls. 488.2 devices also report their status in response to the *STB? query. Consult the device's instruction manual for the meaning of the bits in its Status Register.

In the 488-PC2 Command Set, the status register is serial polled and the value placed in the DevStatus variable by:

```
DevStatus = ieSPoll(DevAddr)
```

For the NI 488.2 Command Set use:

```
Call ReadStatusByte(Bd, Addr, DevStatus)
```

Sending Bus Interface Messages and Addresses to a Device

Sometimes it is necessary to send Bus Interface Messages or Address commands to a device to address a device as a talker or as a listener or to enter a special configuration mode. Interface Messages or Addresses are sent to a device with ATN on. They can be represented by an equivalent ASCII character. Refer to Chapter 1 for a list of these commands.

The 488-PC2 Command Set uses the Interface Message mnemonics. The user specifies the command string and then calls the ieSend command. The ieSend command interprets the mnemonics and converts them into GPIB bus bytes. In the following example, CmdStr$ is set to the escape sequence used with some of ICS's Miniboxes to put them in their command mode. Consult the GPIB

Controller's Command Reference section for the ieSend command mnemonics.

```
CmdStr$ = "UNL LISTEN 4 UNL LISTEN 4 UNL"
L = Len (CmdStr$)
ioerr% = ieSend(DevAddr, CmdStr$, L)
```

In the NI 488.2 Command Set, the Interface Messages and Addresses are sent by specifying the equivalent ASCII characters. i.e.

```
CmdStr$ = Chr$(&H3F) + CHR$(Addr + 32)+ Chr$(&H3F)
     + CHR$(Addr + 32) + Chr$(&H3F)
Call SendCmds(Bd, CmdStr$)
```

Where Addr is the device's address. The NI 488.2 SendCmds command outputs the bytes passed to it in CmdStr$ without the interpretation done by ICS's ieSend command.

Device Addresses

The format of the device address in ICS's command set depends upon the operating system. In DOS, the device address format is *pp* or *pp ss* where *pp* is a primary address and *ss* is a secondary address. Valid addresses are:

0 to 30	for *pp*
100 to 3030	for *pp ss*

In Windows, the device address also includes the card number. The format becomes *cardno pp* or *cardno pp ss*. ICS adopted the Hewlett-Packard convention and numbered the first card as 7. Subsequent cards are numbered 6 down to 4. Valid device addresses are:

700 to 730	for *cardno pp*
70000 to 73030	for *cardno pp ss*

To address a device at primary address 4,

Address = 7 & 04 = 704

To address a device with a primary address of 4 at secondary address 2,

Address = 7 & 04 & 02 = 70402

The NI 488.2 Commands use two variables, *Bd* and *Address* to express the device address. *Bd* contains the Card number and is 0 for the first card. *Address* is a 16 bit variable with the secondary address in the upper byte and the primary address in the lower byte.

Address = [ss + 96]*256 + *pp*

To address a device at primary address 4,

Address = 0 + 4 = 4

To address a device with a primary address of 4 at secondary address 2,

Address = [02+96] *256 + 4
= [98] *256 + 4
= 25088 +4 = 25092

Program Closing

Some GPIB Controller DLLs require that you issue a close command when exiting the program to cleanup the computer's memory and/or to unlock the GPIB controller card for use by another application. Follow the program rules for your GPIB Controller card to end your GPIB program.