TEK GPIB IEEE-488 (1978) PLUS CODES AND FORMATS

# TEKTRONIX CODES AND FORMATS FOR GPIB INSTRUMENTS

GPIB

Tektronix®
COMMITTED TO EXCELLENCE

## Contents

As their measurement needs have grown in number and complexity, many instrument users have realized that their traditional design and test procedures are inadequate. They have new needs which can be satisfied by assembling their individual instruments into interactive and automated systems that will:

- Reduce labor costs.
- Increase the effective use of research and design skills by freeing them for creative work.
- Provide insight into products and processes through coupling analysis with measurements.
- Reduce human errors with precise and repeatable measurements.

Meeting these needs requires instruments, controllers, and peripherals which are easy-to-use with each other.

The first major step toward device compatibility was taken in 1975 when the IEEE published the 488 standard defining an interface for programmable instruments. This bus is usually called the GPIB—General Purpose Interface Bus. Before GPIB, connecting programmable instruments to a computer or to a desktop calculator was a major job because each instrument's interface was different. Now, the IEEE Standard 488 defines an interface that makes it much easier to put together computer-controlled instrument systems.

The IEEE-488 standard defines three aspects of an instrument's interface:

1. Mechanical—the connector and the cable.
2. Electrical—the electrical levels for logical signals and how the signals are sent and received.
3. Functional—the tasks that an instrument's interface is to perform, such as sending data, receiving data, triggering the instrument, etc.

Using this interface standard, instruments can be designed for base compatibility. However, it is only the *first* step toward further standardization for even greater compatibility.

Tektronix has taken the next step by adopting a new standard called Codes and Formats. It is intended to:

- Define device-dependent message formats and codings and thus enhance compatibility among instruments that comply with IEEE Standard 488-1978,
- Reduce the cost and time required to develop system and applications software by making it easier for people to generate and understand the necessary device-dependent coding.

Beyond the Codes and Formats standard, there is also a need for a philosophy of designing instruments to be friendly to the user. They should be controlled over the bus with easily understood commands and should be resistant to operator errors. Since the application of this philosophy is different for each type of instrument, it is not included as a specific standard.

This document explains the details of, and reasons for, the Tektronix Codes and Formats standard; as well as the concept of instruments designed for people.

### Setup for Automated Measurement Analysis

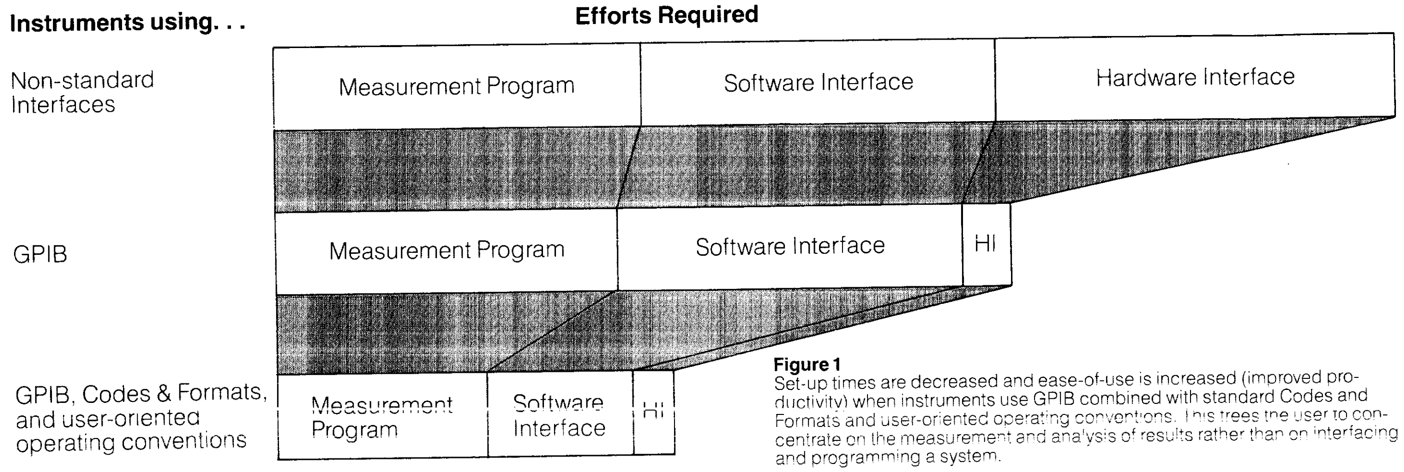


**Figure 1**
Set-up times are decreased and ease-of-use is increased (improved productivity) when instruments use GPIB combined with standard Codes and Formats and user-oriented operating conventions. This frees the user to concentrate on the measurement and analysis of results rather than on interfacing and programming a system.

# TEKTRONIX GPIB IMPLEMENTATION

## Compatibility plus Capability

A key factor in determining the productivity of an instrument is its user interface. Previously most interfaces were designed with the instrument logic being the prime concern. Ease-of-use was often sacrificed to keep the instrument hardware simpler (and less costly). Today, with the abundance of low-cost, powerful microprocessors, it is possible to design instruments with ease-of-use as a primary objective without adding significant cost or degrading performance or GPIB compatibility.

Tektronix GPIB instruments demonstrate this crucial difference between GPIB compatibility and capability. They are programmable instruments designed with easily understood commands and with standard data transfer conventions defined by Tektronix Codes and Formats. This can increase your productivity by improving the usability of your instruments.

Using the General Purpose Interface Bus is somewhat like using the telephone system. In both cases, a physical connection can be established between two locations and data can be transmitted—i.e., one person, or instrument, can talk to another. However, on the telephone system, unless both people speak and understand the same language, very little communication can take place. Beyond having a common language, they must also share the same vocabulary for real communication to take place. Similar problems can arise between different instruments not specifically designed to work with others. A common protocol, or "telephone manners," is important, as well—one person should not hang up before the other has finished speaking and has said "goodbye."

The IEEE-488 standard defines a "telephone system" describing how the physical communications system is to be used, but it does not define the "language" sent over the bus or the "manner" in which the physical communications system is to be used. This lack causes incompatibilities. For example, assume that a multimeter has made a measurement of +3.75 volts and now has to transmit this information over the GPIB to a computer. Eight data lines are available for sending information in a byte—serial fashion. The first question is: "What codes should be used to encode the five characters?" The 488 standard gives no recommendations here. The DMM designer is free to choose any code he wishes. If a BCD code is selected but the computer only understands ASCII, the multimeter and computer will be incompatible when connected, even though both devices are "standard."

Suppose the multimeter above does send ASCII code? The question now is, "What Format is the data to be in?" Do we send the character sequence +3.75 right to left, left to right, or some other way? Again, the IEEE-488 standard says nothing regarding the sequence of data bytes. The instrument designer is free to create incompatibility. Figure 2 illustrates three formats for the data from a DMM. For system products, it is important that designers use a common format. Thus, the Tektronix Codes and Formats Standard specifies that ASCII data be sent with the most significant data first, thus reducing the options for instrument designers, and making it easier for users to incorporate instruments into systems.

As more instruments incorporate microprocessors, increase in complexity, and gain intelligence, the potential for incompatibility will become more acute. What is needed is a standard to bring data compatibility to the GPIB in the same way that the IEEE-488 standard brought electrical compatibility to instrument interfaces. Tektronix has adopted such a standard—and has incorporated it in the design of all Tektronix GPIB instruments.

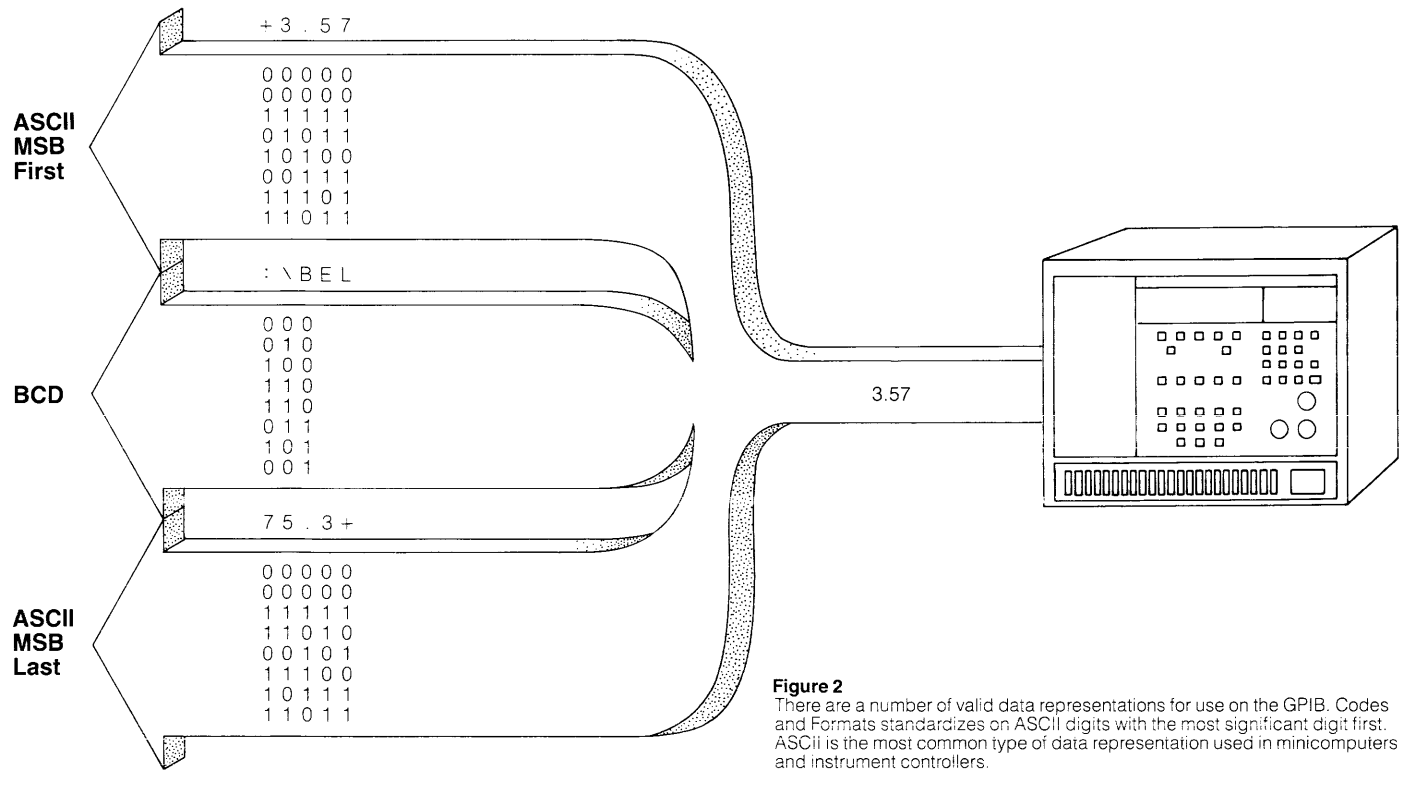


**Figure 2**
There are a number of valid data representations for use on the GPIB. Codes and Formats standardizes on ASCII digits with the most significant digit first. ASCII is the most common type of data representation used in minicomputers and instrument controllers.

# The Tektronix Codes and Formats Standard

This standard:

- establishes a common message structure.
- describes communication elements and how they may be combined.
- defines control protocol.
- standardizes features that are particularly important to test, measurement and analysis systems.

System developers incorporating instruments which comply with the Codes and Formats standard as well as using easily understood commands will find that:

- writing the system software will be easier.
- with a defined language syntax, programs will be more efficient and self-documenting, making software maintenance easier.
- system change, or expansion, will require less software modification; and many support subroutines, such as error handling, won't need to be changed.

Also, Codes and Formats are important from the instrument's point of view. It's relatively easy to tell a person that a particular number format is to be used; most people intuitively understand what a number is.

A microprocessor in an instrument, however, has no such intuition and must be told—explicitly and unambiguously—how to send or how to receive numbers. Otherwise, it can send or receive something that is "obviously" wrong.

Because nearly all of today's GPIB instruments use ASCII-coded characters to send and receive data, Tektronix has chosen ASCII coding as standard.

In addition, nearly all instruments that send or receive numbers use the ANSI X3.42 standard format. This format states in effect that there are three types of numbers—integers, reals, and reals with exponents—and that they should be sent with the most significant character first. Table 1 shows examples of these formats. Unless there are numeric needs that cannot be met by this standard format, present and future Tektronix instruments will also use this format.

Note however, that while a number has been defined, its use has not. The Codes and Formats Standard has placed no restrictions upon the use of the number. It does not matter whether the number is sent from a multimeter, a counter, or a spectrum analyzer. In all cases, the syntax or structure of the number is identical. The semantics or meaning may be as different as VOLTS, PARTICLES, or CENTER FREQUENCY. Having this well-defined format for using a number allows instruments to send and receive numbers without confusion. Clearly, this is a step towards "language" compatibility.

As more instruments incorporate microprocessors, the functions they can perform will become increasingly complex. To have a computer or instrument controller interact with such intelligent devices requires Code and Format conventions more comprehensive than those that simply define numbers.

For instance, suppose that a device makes a group of measurements and is asked to report them. This requires that a group of numbers be sent. To separate one number from the next, a comma is used. For example, the position coordinates from a digitizer might be sent as .732, 1.52.

Furthermore, suppose that another device makes two different types of measurements and is required to report them—for example, FREQUENCY and PHASE. There should be a means for identifying each type. This is to be done by first sending a "header," that is a description of the number. If these headers and numbers are sent consecutively, they must be separated from each other. For this, a semicolon is used. For example: FREQ 1234; PHASE 56 or FREQ 1.234E03; PHASE 56.

These well-defined formats significantly enhance data communication compatibility over the GPIB.

### TABLE I

**NUMBER FORMATS (ANSI X3.42)**

| | | |
|---|---|---|
| NR1 | 375<br>+8960<br>−328<br>+0000 | Value of "0" must not contain a minus sign. |
| NR2 | +12.589<br>1.37592<br>−00037.5<br>0.000 | Radix point should be preceded by at least one digit.<br>Value of "0" must not contain a minus sign. |
| NR3 | −1.51E + 03<br>+51.2E − 07<br>+00.0E + 00 | Value of "0" must contain a NR2 zero followed by a zero exponent. |

## The Human Interface

The shape of current and future technology requires people with a wide range of technical skill levels to be intimately involved in the instrument-to-instrument communication process. For example, if a GPIB-programmable power supply needs to be set to 20.0 volts, a person has to write this "need" into the controlling computer's program. The computer becomes an intermediary transmitting the person's intent over the bus to the power supply.

The power supply can be designed in one of two basic ways. The first is with minimal intelligence so that it can accept some "hieroglyphics"—which it in turn can conveniently interpret and execute. For example, some power supplies require the sequence 0 8 E 3 to put out 20 volts. Here the "0" stands for the 0 = 36 volt range, and the "8E3" is the ASCII representation of the hexadecimal commands required as shown in Figure 3.

On the other hand, the power supply can be designed with a microprocessor and intelligence to accept easily understood numbers. In this example, to put out 20 volts, the programmer simply sends the character sequence "VPOS 20." This second method of interacting is obviously a great deal more convenient for people, not only when the computer program is first written but also later, when someone other than the original programmer has to find out what the program is supposed to do. In the future, most devices will be "intelligent" and specifically be designed to interact with people. In order to promote this compatibility, the appropriate Codes and Formats are used.

Numbers in easily read formats are good for both computers and people. It is also necessary to send directions to instruments in a format other than numbers. For example:

TRIG EXT
CLIP ON
PEAK AUTO
FUNC SINE

In these situations, we can simply treat the first word as a header and the second word as a data type that is different from a number.

Other data, called arguments, are useful for various purposes:

String Arguments—for sending text to a display or printer.

Binary Block Arguments—for sending binary data blocks of known length.

Link Arguments—for sending certain types of instrument commands.

End Block—for sending binary data of unknown length or format.

The same general format can be used for all types of communications over the bus—commands to instruments, data from instruments, text to be displayed, and more. This message structure is summarized in Table II.

In basic terms, this is the Codes and Formats Standard adopted by Tektronix. It extends instrument compatibility from the realm of the electrical signals defined in IEEE 488 to the domain of data sent over the GPIB. It extends not only to computers and simple and smart instruments but also to people of widely divergent technical skill levels who have to work with these devices.

To represent the 20 Volts we want from the 36 Volt power supply let the calculator first compute

$$20/36 = V/4095$$
$$V = (20 * 4095)/36 = 2275$$

Then we need a subroutine to convert 2275 (base 10) to its hexadecimal equivalent (base 16).

The way the calculator will do this is to divide by 16 until the remainder is less than the base 16, each time converting the fractional part of the remainder into a whole number by multiplying the fraction by the base, 16.

$$2275/16 = 142.1875$$

To get an integer remainder you multiply:

$$.1875 * 16 = 3 \text{ (3rd character)}$$
$$142/16 = 18.875$$

Multiply

$$.875 * 16 = 14 \text{ (2nd character)}$$

8 is less than 16, therefore there is no further division and 8 is the 1st character.

In hex, you would write 8 14 3 as 8E3.

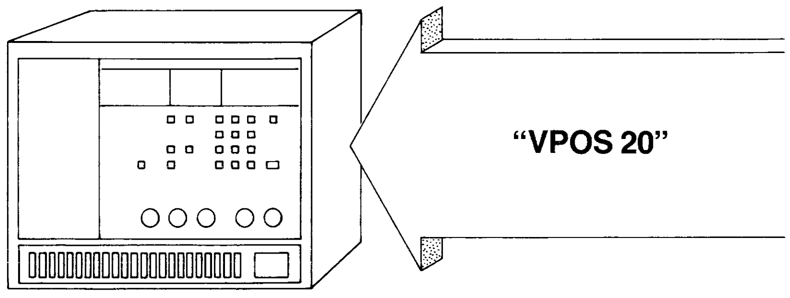


**Figure 3**
The use of microprocessors in the design of "intelligent" instruments allows instrument commands which are oriented to the user. This allows users to program instruments with understandable commands such as VPOS 20 to set a power supply to a positive 20 volts.

## TABLE II

## DEVICE-DEPENDENT MESSAGE STRUCTURE

A message represents a given amount of information whose beginning and end is defined. It is communicated between a device functioning as a talker and one or more devices functioning as listeners.

A message begins when the transmitting device is initially addressed to talk and the receiving device(s) is (are) addressed as listener(s).

A message is composed of one or more message units separated by message unit delimiters. A message unit delimiter is a semicolon.

The message ends when the talking device asserts EOI.

**There are two message unit types:**

1. Mixed Data Message Unit in two acceptable formats are:
   a. Header (in Character Argument format) followed by a space and optional arguments of any type separated by commas. Normally used for programming information.
   b. Non-character argument followed by optional arguments of any type separated by commas. Normally used for measurement data.
2. Query Message Unit consists of a character argument such as "SET," "ID," or "FREQ" followed by a question mark. This is normally used to interrogate a device for data or settings.

**Following are definitions of the allowable argument types:**

| **ARGUMENT TYPES AND EXAMPLES** | **DEFINITION** | **PURPOSE** |
|---|---|---|
| Character Argument<br>Trigger | One alphabetic ASCII character optionally followed by any number of ASCII characters excluding space, comma, semicolon, question mark, the control characters and rubout. | Used to transfer alphanumeric data such as message headers, labels, commands, etc. |
| Non-Character Arguments | | |
| Number<br>−12.3 | A numeric value in any of the formats shown in Table 1. | Used to pass numeric values in an ASCII format. |
| String Argument<br>"Remove Probe" | Opening delimiter (single or double quote) followed by a series of any ASCII characters except for the opening delimiter and a closing delimiter identical to the opening delimiter. | Provides a means for transmitting ASCII text to an output device. |
| Binary Block Argument<br>$\%\ \underbrace{\text{2 bytes}}_{\text{16 bit binary value}}\ \overbrace{\text{wave-form values}}^{\text{Binary Data}}\ \underbrace{\text{8 bits}}_{\text{checksum}}$ | "%" followed by a 2 byte (16 bit) binary integer specifying the number of data bytes plus a checksum byte which follows the data bytes. The checksum is two's complement of the modulo 256 sum of the preceding binary data bytes. This includes the two bytes comprising the 16 bit integer specification. | Used to transfer large arrays of numeric data such as waveforms in a binary format. |
| Link Argument<br>NR.PT:1024 | Character argument (label) followed by ":" and a value represented in any of the above argument types. | Used to attach a name or label to another argument. |
| End Block Argument<br>@ABCDEFGHIJKL $\begin{bmatrix}E\\O\\I\end{bmatrix}$ | "@" followed by a block of data with EOI set concurrent with the last data byte. End block can only be the last argument in a message and cannot be followed by a message unit delimiter. | Used when a block of data must be sent and neither the amount of data nor its format is known. |

## Message Conventions

While standardizing this Codes and Formats "language" fosters greater compatibility between devices using the GPIB, it alone does not solve all compatibility problems. Well-defined operational conventions are also needed.

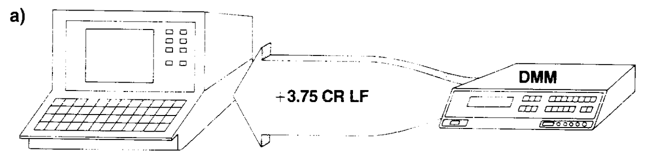
Conventions for using the GPIB are analogous to good manners when using the telephone: one party should not hang up before the other has finished speaking. The following is an example of the lack of good manners in two devices using the GPIB.

Suppose a computer has requested a voltage reading from a multimeter over the GPIB. The multimeter sends the number and terminates the transmission with the characters CR (carriage return) followed by LF (line feed) (Figure 4a). The computer, however, understands that CR by itself terminates a message. The computer "hangs up" after receiving the CR and leaves the LF character in the multimeter unsent (Figure 4b). The next time the computer asks for a multimeter reading, the multimeter sends LF, the character left over from the previous measurement, followed by the measured values (Figure 4c). The computer does not know what to make of a number preceded by a LF character. It stops and indicates an error. Although both devices are GPIB compatible, they do not work together because the IEEE 488 standard has not defined the conventions, "manners," for how the bus is to be used.

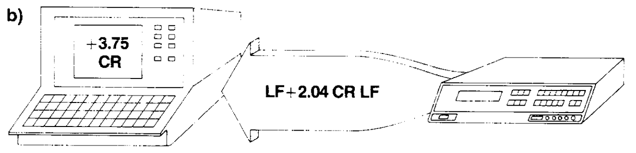
To avoid such incompatibilities, a standard way to terminate messages is needed. Two methods are commonly used. The first is to send some printer format characters such as CR or CR LF. The other is to assert the EOI line when the last data byte of a message is sent. The first method was adequate for simple instruments that sent or received only ASCII coded numbers. However, today's more intelligent devices have to send messages representing digitized waveforms or programs for a microprocessor in an instrument. Some of these messages may contain binary data to reduce transfer time. A certain sequence of binary coded bytes (00001101, 00001010), which if interpreted as ASCII, will appear to be a CR LF, and thus be misinterpreted (Figure 5). The second termination method has no such problems. Asserting the EOI line, unambiguously terminates the message.

Thus, the Tektronix Codes and Formats Standard states that instruments sending messages should terminate them by asserting the EOI line concurrently with the last byte of the message (Figure 6).

a)



b)



c)

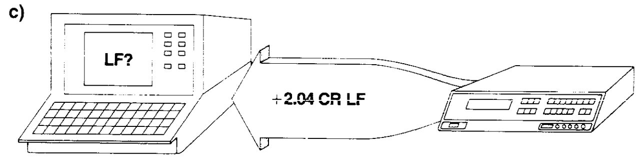
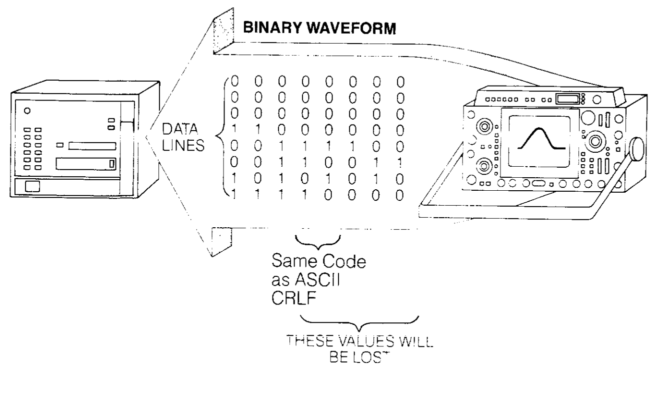


**Figure 4**
Without a standard message termination, the user must take care that each listener understands the same message termination convention. In this example, (a) the talking instrument uses CR LF as the message terminator, (b) while the listening controller understands CR as the message terminator. (c) When the controller accepts the next message, the first character encountered is the LF left from the last message which the controller may interrupt as an illegal character since it is expecting a numeric value.



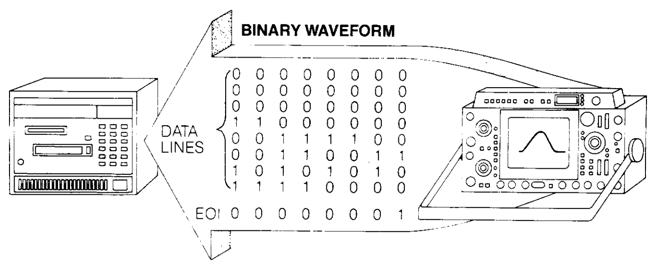


**Figure 5**
When large amounts of data (e.g., a waveform) are to be transferred over the GPIB, it is often more efficient to code this data in binary rather than ASCII. This can provide more than a 2:1 reduction in the number of bytes to be transferred. However, if CR LF is used by the listener as a terminator, and the binary data coincidentally has the binary equivalents of these characters, the listener will stop listening in the middle of the transmission. Thus, EOI is a better choice for an unambiguous message termination convention.

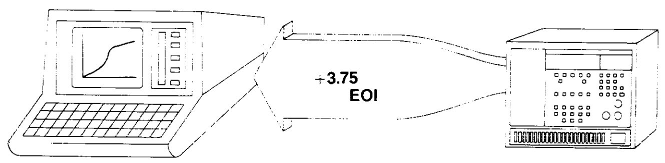


**Figure 6**
Setting the EOI line concurrent with the last byte of a message provides an unambiguous message terminating convention.

Other problems can be created by instruments which execute each individual command as received. For example, suppose a programmable high-voltage power supply that can be set as high as 1000V has been set to put out 10V and limit current at 2A. Then it is sent the message "VOLTS 1000, CURR 10E – 06," that is, 1000 volts output limited at 10 MA. Lacking good message-handling conventions, the supply goes to 1000V output immediately upon receiving the first part of the message. But because the current limit is still 2A, the value from the previous setting, the supply either crowbars or damages the equipment connected to it (Figure 7). For proper operation, the programmer should have changed the current setting first. Only then should the voltage be changed. It is much easier and safer using a power supply which does not execute any command until the entire message is received and terminated by asserting the EOI line (Figure 8).

This same convention can also prevent misunderstandings between a computer and a measurement instrument. When instructed to send a measurement message, the instrument sends EOI only when the message is complete, and no more data is sent unless directed by the computer to do so. This way the computer knows that no data is lost; the instrument is not inadvertently stopped from talking in the middle of a message.

In short, Tektronix Codes and Formats standard defines a message to be a complete block of information. It begins when a device starts sending data and ends when EOI is sent or received concurrently with the last data byte.

It should be noted, too, that the beginning of a message will need further clarification. An instrument sending a message may be interrupted by the computer taking control, perhaps conducting a serial poll. When the instrument becomes a talker again, it should resume sending the message. Thus, a message begins when a device enters the talker active state for the first time following a reset or a previously sent EOI.

There is a further refinement to the message convention. When a device is made a talker, it should always say something. If it has nothing to say and will have nothing to say for an indefinite period of time, it should send a byte of all ones concurrent with EOI. This lets the listening device know that no meaningful data is forthcoming. Thus, the "talked with nothing to say" byte is a null message. This convention prevents tying up the GPIB while the computer waits for a device to talk that will never send a message.

There are other conventions associated with messages that make life on the bus easier. A listening device should always handshake. It should not stop handshaking just because it does not understand or cannot execute a particular message. After EOI is received, if the listening device is confused, it should send out a service request and, on a serial poll, notify the controller that nonsense has been received. *Under no circumstances should a device execute a message it does not understand.* Some non-Tektronix devices do not follow this convention—with disastrous results. A particular power supply can be sent four letter O's instead of four zeros, a common human mistake, and this supply will put out its maximum voltage instead of the intended zero volts.

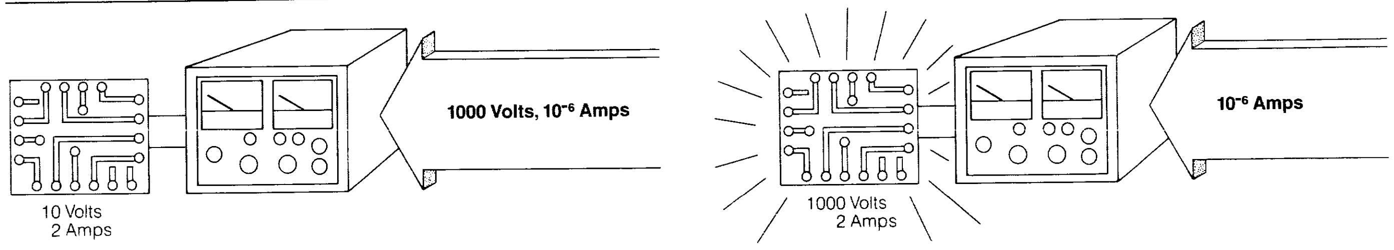


**Figure 7**
Instruments should not begin execution of any part of a message until the entire message has been accepted. In this example, the settings of a general purpose power supply are to be changed from 10 volts, 2 amps to 1000 volts, 1 milliamp. If the voltage command is executed prior to changing the current setting, the results could be disastrous.

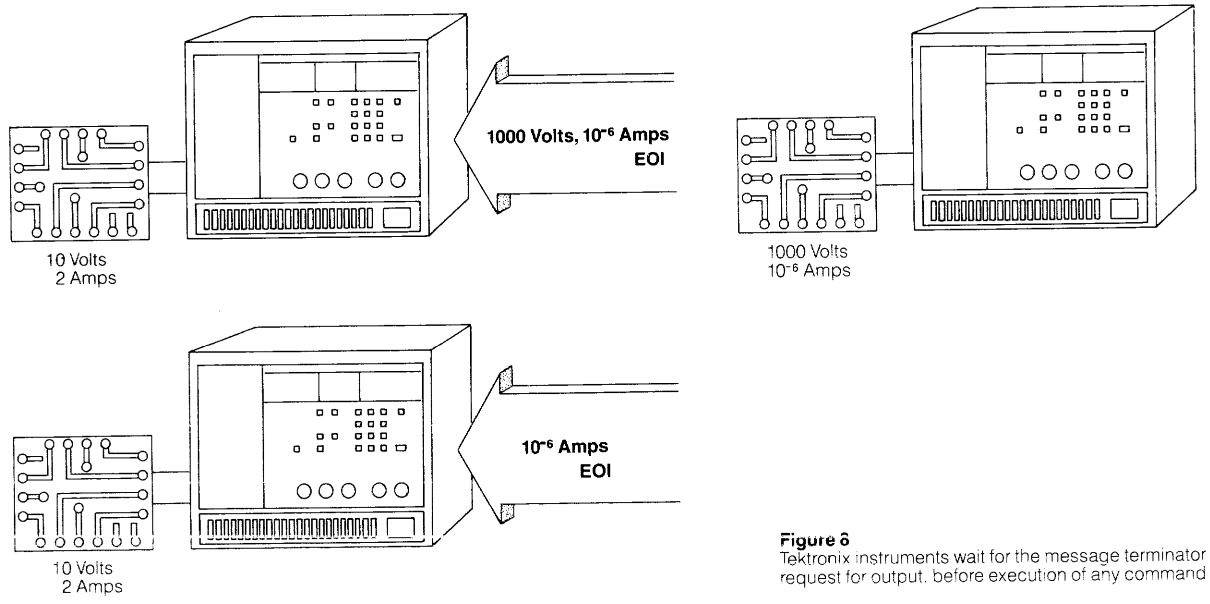


**Figure 8**
Tektronix instruments wait for the message terminator (EOI), or optionally a request for output, before execution of any command in the sequence.

# Status Bytes

The IEEE 488 standard defines a facility for an instrument to send a byte of status data to the computer, but, except for bit 7, the standard does not define the meaning of the bits. The IEEE 488 standard assigns bit 7 to mean that a device is, or is not, requesting service. Thus, bit 7 cannot be used for other purposes.

However, there is a common need for instruments to report certain kinds of status or errors to the computer. So a status byte convention is established for this. One common need is for instruments to report if they are busy or ready. Bit 5 is used for this purpose. Another common need is for instruments to report if they are encountering abnormal conditions. Bit 6 is selected for this.

There are more complex conditions besides busy/ready or normal/abnormal. These are listed in Table III. While these status bytes are generally useful for most purposes, certain instruments may have conditions that are peculiar to them. To report these status states, bit 8 is used to indicate that the status byte is not the common type but particular to an instrument.

Providing a standard coding for the status byte enhances the convenience to the person programming the system. If all the instruments have common status byte codings, then a common status byte handling routine is written for all instruments, not a separate one for each.

## TABLE III STATUS-BYTE DEFINITIONS

| | | Decimal | |
|---|---|---|---|
| **Abnormal Conditions** | **Binary** | **X = 0** | **X = 1** |
| ERR query requested | 011X 0000 | 96 | 112 |
| Command error | 011X 0001 | 97 | 113 |
| Execution error | 011X 0010 | 98 | 114 |
| Internal error | 011X 0011 | 99 | 115 |
| Power fail | 011X 0100 | 100 | 116 |
| Execution error warning | 011X 0101 | 101 | 117 |
| Internal error warning | 011X 0110 | 102 | 118 |
| **Normal Conditions** | | | |
| No status to report out of the ordinary | 000X 0000 | 0 | 16 |
| SRQ query request | 010X 0000 | 64 | 80 |
| Power on | 010X 0001 | 65 | 81 |
| Operation complete | 010X 0010 | 66 | 82 |

## Definition of the system status Bytes.

**ERR query request**—This states that a device is reporting an error but is not identifying the error. The controller should send ERR? to find out about the error.

**Command error**—Results when a message can't be parsed or lexically analyzed.

**Execution error**—Exists when a message is parsed and analyzed but can't be executed (e.g., setting a device out of its range).

**Internal error**—Results from an out of calibration error or a malfunction in the device (may be discovered by an instrument self-check).

**Power fail**—Some instruments can detect a power fail long before the instrument is affected. In those cases a power fail is sent to the controller to save important information or to flag suspect operation.

**Execution error warning**—The device indicates that it has received and is executing a command but that a potential problem might exist. For example, an instrument may be out of range but is sending a reading anyway.

**Internal error warning**—The device indicates that it has an internal error but is continuing to function.

**No status to report out of the ordinary**—When the controller does a serial poll and a device has nothing to say, this byte is reported.

**SRQ query request**—This is similar to the ERR query request. It simply says "I am requesting service but am not saying why." To find out why, the controller should send SRQ?.

**Power on**—When a device has finished its "power on" sequence, it may send a "power on" service request. This will let the controller know that a device has just come up on the bus. It should be noted that on a "power on" service request, RSV can not be cleared by a device clear. The only thing that can clear the RSV is handshaking out the status byte to the controller. This eliminates the problem of a device powering up and receiving a device clear before the controller knows it is on the bus. After every power on, this status byte should be the first reported.

**Operation complete**—This tells the controller some task has been completed. This is not the same thing as the BUSY bit, DIO5, described earlier. The BUSY bit going false indicates that processing and execution of a command has been completed. In a multi-task environment, the execution of a command may cause a long task to start, e.g., data log 10,000 points. At this point the device is ready for another command. When the task is done, the "operation complete" status byte is sent to the controller.

## Queries

Even with all the possibilities allowed by status bytes, it is often necessary to send more detailed information from an instrument to a computer. This can be done via "queries."

Normally a measurement instrument sends a measured value when it functions as a talker. To elicit something other than this default message, the computer can first send a query to the instrument. Then, when the instrument is made a talker, it will send the desired information. Queries take the form of a Header followed by a question mark. An example is shown in Figure 9. Here are some queries and their uses:

ERR? is used for investigating detailed error conditions in an instrument. The response an instrument sends back is ERR followed by NR1 numbers that code the particular problem.

SET? requests an instrument to send the computer its present settings and other current state information. Sending this information back to the instrument at a later time returns the instrument to the state it was in when it received SET? This query makes it possible to develop a program using an instrument's front panel as an input to the computer. Using this feature, a programmer never needs to know the instrument's GPIB commands.

ID? makes an instrument identify itself by sending information as instrument type, model number, firmware version, etc. This feature is useful for identifying a particular device in the field and potentially for self-configuring systems.

Defining a standard way to elicit responses from an instrument enhances the convenience to the programmer. When all instruments use the same form to perform similar functions, the programmer has to learn only one convention, not many.

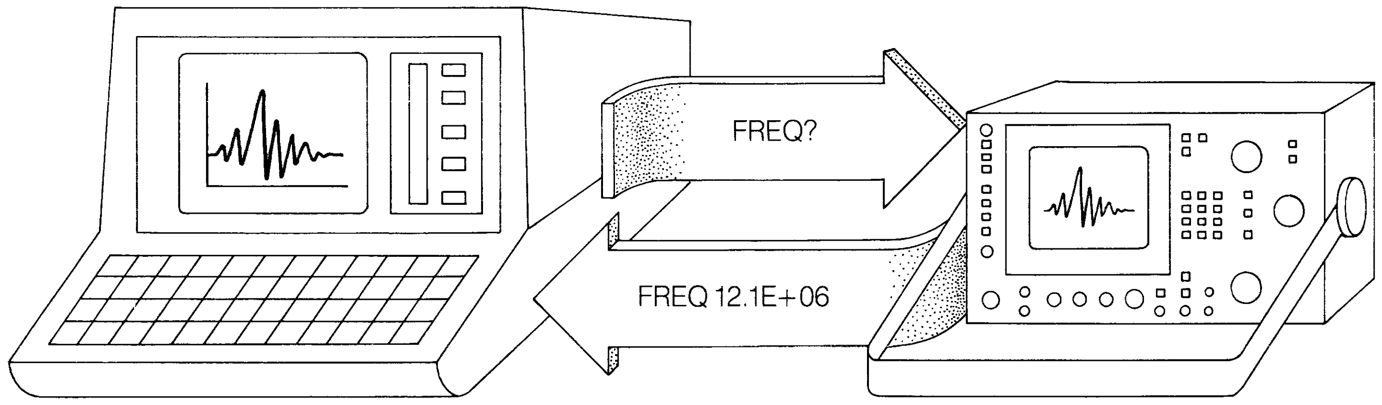


**Figure 9**
The use of easy-to-remember queries is an important feature of Tektronix GPIB instruments. Many query commands are formed by adding a question mark to the mnemonic for the setting to be queried.

## Capabilities and Complexities

Instruments that incorporate queries are more compatible and can be more useful in systems. As instruments get more complex, and in many ways become computers in their own right, more operational conventions are needed. These conventions are more characteristic of communication between a computer and its peripheral processors than what we see today between small desktop calculators and simpleminded instruments. But such conventions need not make intelligent instruments more difficult to use. Quite the contrary, an intelligent instrument should be easier to use if its intelligence is used properly. Here are some examples of conventions adopted by Tektronix to enhance both computer/instrument compatibility and human/system compatibility.

1. While an instrument should always send numbers in the correct format described earlier, it should receive numbers "forgivingly." Specifically:
   a. Negative zero numbers should never be sent, but they should be accepted.
   b. Any number in scientific notation should be sent exactly as defined in ANSI X3.42 standard, i.e., with the decimal point included. Some of today's computers violate this standard and send values without the decimal pont.

      This "illegal" number should be received with an implied decimal point following the least significant digit.
   c. If an instrument receives a number whose precision is greater than the instrument can handle internally, then the number should be rounded off, not truncated. This enhances instrument accuracy.
2. An instrument should recognize both spaces and commas as argument delimiters. Multiple spaces or commas should not be construed as delimiters for null arguments. This convention is useful because some computers gratuitously generate spaces and send them on the GPIB.
3. An instrument should receive headers and character arguments in both upper and lower case and equate them. a=A, b=B, etc. This is useful because some desktop instrument controllers have a problem sending upper-case alpha characters.
4. An instrument sending data about its front panel should use headers and character arguments that correspond to the front panel's nomenclature.

These features make Tektronix instruments "friendly" to a casual or inexperienced programmer and compatible with most computers. There are still other features that are built into Tektronix intelligent instruments. Here are two examples:

The Service Request (SR) function and corresponding status byte reporting are very important. They can alert the controller of programmable instruments to new events or possible malfunctions (Figure 10). Thus, they let the computer controlling an unattended instrument system manage the system effectively or call for help when it can't. Sometimes, however, a person or a computer does not want to be interrupted—for example, when a sensitive or time-dependent measurement is being made. For these cases a message, RQS OFF, can be sent to disable any service requests. To turn the service request capability back on, the message RQS ON is sent.

Another useful convention is related to the Device Trigger (DT) function. Sometimes a command message sent to an instrument should be executed immediately. At other times, the command message should only set up the instrument, and the desired action should be executed when the Group Execute Trigger interface message is sent. To make the instrument execute commands immediately, the message DT OFF is sent. To make the instrument defer execution of commands, the message DT ON is sent.

These and other features of intelligent instruments make them both compatible with computers and friendly to people. Compatibility and friendliness are really the same thing. Given a powerful enough computer, and a clever programmer, most of today's devices that use the IEEE 488 bus can be made to do whatever they were designed to do: compatibility can be forced. Without well-defined codes and formats and without operational conventions and easily understood commands, instruments appear incompatible or unfriendly. With codes and formats and with known operational conventions, devices using the GPIB become friendly as well as compatible, thereby allowing the user to spend more time on the task at hand, rather than figuring out how to make the system work—increased productivity with Tektronix GPIB instruments.

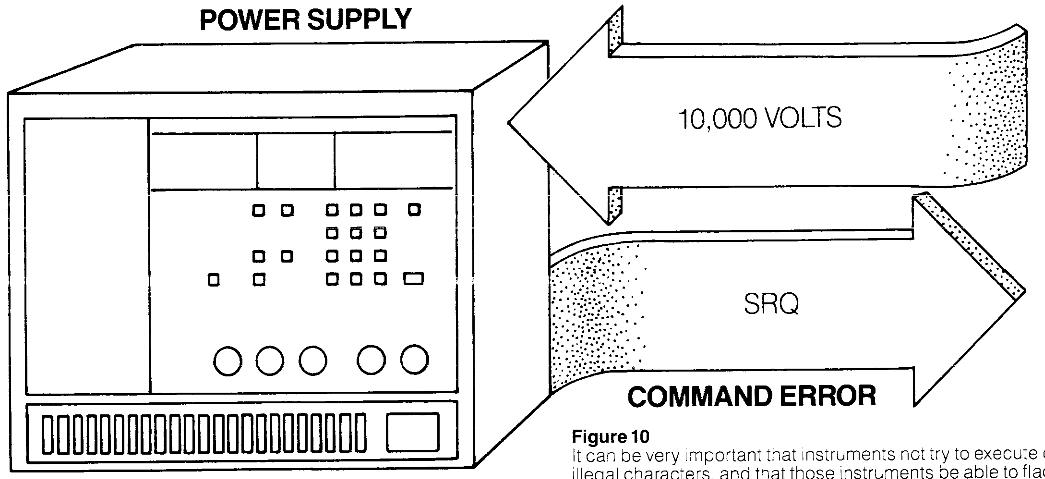


**Figure 10**
It can be very important that instruments not try to execute commands with illegal characters, and that those instruments be able to flag the controller that an error has occurred. In this case, a typing error may have created an illegal command which some power supplies would interpret as their maximum setting. This should not be allowed to happen!

For further information, contact:

**U.S.A., Asia, Australia, Central & South America, Japan**
Tektronix, Inc.
P.O. Box 4828
Portland, OR 97208
Phone: 800/547-6711
Oregon only 800/452-6773
Telex: 910-467-8708
Cable: TEKTRONIX

**Europe, Africa, Middle East**
Tektronix International, Inc.
European Marketing Centre
Postbox 827
1180 AV Amstelveen
The Netherlands
Telex: 18312

**Canada**
Tektronix Canada Inc.
P.O. Box 6500
Barrie. Ontario L4M 4V3
Phone: 705/737-2700

**Tektronix sales and service offices around the world:**
Argentina, Australia, Austria, Belgium, Bolivia, Brazil, Canada, Chile, Colombia, Costa Rica, Denmark, East Africa, Ecuador, Egypt, El Salvador, Federal Republic of Germany, Finland, France, Greece, Hong Kong, Iceland, India, Indonesia, Iraq, Israel, Italy, Ivory Coast, Japan, Jordan, Korea, Kuwait, Lebanon, Malaysia, Mexico, Morocco, The Netherlands, New Zealand, Norway, Pakistan, Panama, Peru, Philippines, Portugal, Republic of South Africa, Saudi Arabia, Singapore, Spain, Sri Lanka, Sudan, Surinam, Sweden, Switzerland, Syria, Taiwan, Thailand, Turkey, Tunisia, United Kingdom, Uruguay, Venezuela, Zambia.

Tektronix®
COMMITTED TO EXCELLENCE