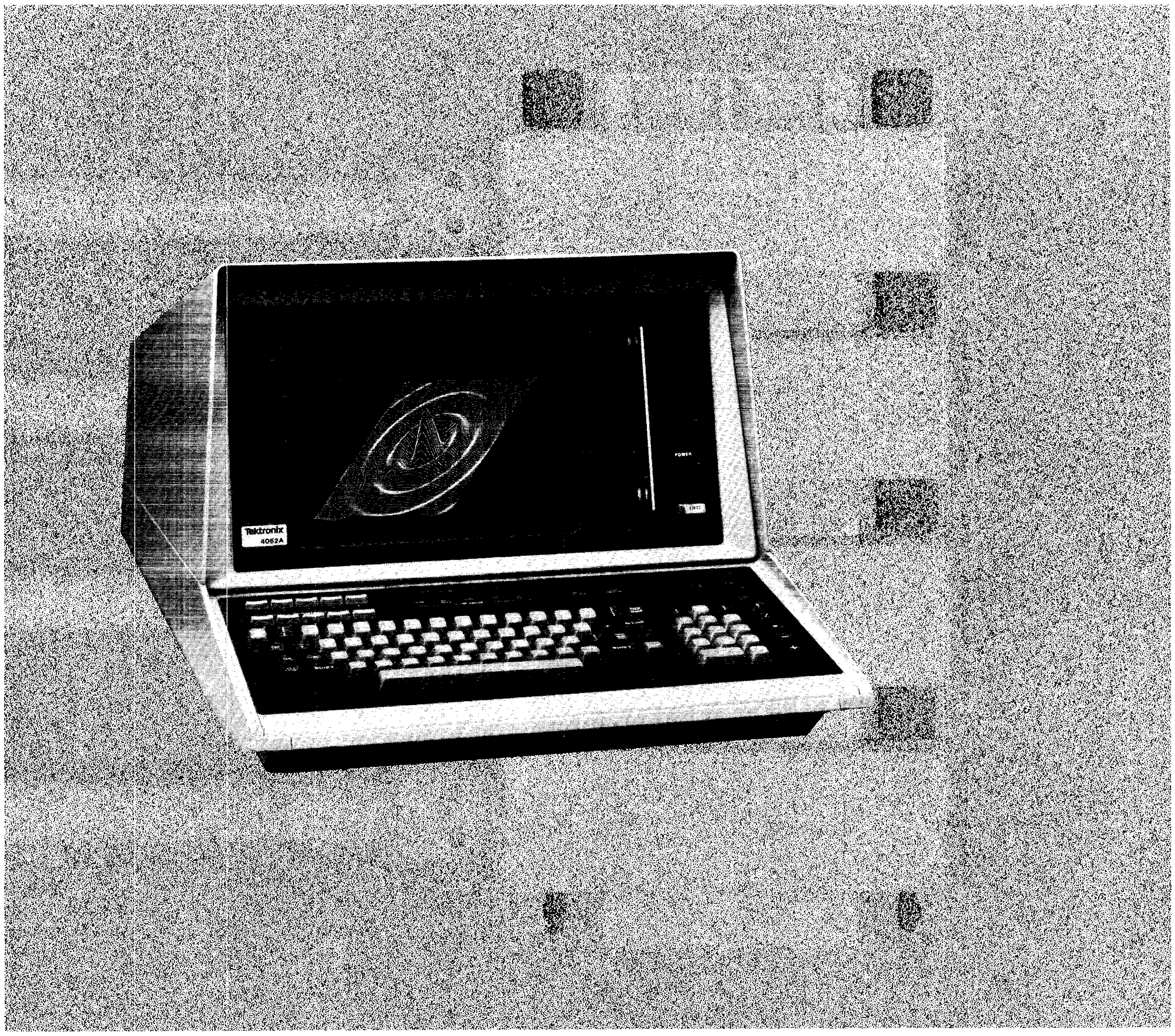
TEK PROGRAMMING INFORMATION FOR IEEE 488 CONTROLLERS

GPIB MADE EASY

4052A GPIB Programming Guide



Tektronix®
COMMITTED TO EXCELLENCE

A binder is available from Tektronix for your GPIB Programming Guides. Contact your local Field Office or representative and ask for part number 062-6433-00.

Additional Application and Programming Resources from Tektronix

- Application Engineers at many local field offices
- HANDSHAKE—Newsletter of Signal Processing and Instrument Control
- Tektronix Instrumentation Software Library
- Application Notes
- Other GPIB Programming Guides
- Instrument Interfacing Guides (IIGs)
- Utility Software for programmable instruments

For more information, contact your local Tektronix Field Office or representative.

This Programming Guide was written by Harold Mendoza and produced by the ISD System Support Group.

070-5012-00

First Printing NOV 1983

Preface

What This Programming Guide Is

The GPIB can be a smooth path to automated test and measurement, or it can be a rough road, strewn with pitfalls. Choosing the right controller and instruments and writing efficient control programs can make the difference. This programming guide provides some guidelines for selecting system components and implementing a system based on the 4052A Desktop Computer.

Section 1 is a brief introduction to 4052A GPIB capabilities.

Section 2 discusses guidelines for choosing system components and configuring the system.

Section 3 reviews the fundamentals of 4050 BASIC.

Section 4 gets down to the specifics of GPIB system programming with the 4052A.

Section 5 is devoted to techniques for processing and displaying acquired data.

Section 6 describes the factors that affect system performance.

Section 7 provides some hints for improving system performance.

Section 8 describes how to make GPIB programming easier with the 4052R14 GPIB Enhancement ROM pack.

Though this guide is for the 4052A, it also applies to the 4054A, and much of it applies to the 4051. The 4052 has its own programming guide that also applies to the 4054. Specific language differences for the 4051, 4052, 4052A, 4054 and 4054A are contained in the 4050 Series Graphic System Reference Manual.

What This Programming Guide Is Not

This programming guide is not a GPIB reference book—that function is provided by the IEEE standard 488-1978. Neither is it 4052A reference manual or an instrument reference manual—these books already exist.

Rather, this programming guide is a practical approach to a 4052A-controlled GPIB system with only enough reference information and theory to enable you to configure and understand a GPIB system to fit your needs.

Table of Contents

Section 1 — The 4052A as a GPIB Controller

Defining the System Controller's Job

A typical GPIB system (Fig. 1-1) could include a controller, such as the Tektronix 4052A Desktop Computer, a signal generator only able to listen, a digital counter, able to talk and listen, and a magnetic tape drive, able to talk and listen. These instruments can work together to perform a task, but they must be directed—and that's where the controller comes in.

At the heart of the GPIB system is its controller. In all but the simplest data-logging applications, some form of controller is required to make the system work. But, taking full advantage of the controller's power requires a good understanding of its job in the GPIB system. The controller's job can be broken into four major tasks:

1. Program development
2. Instrument control
3. Data processing
4. Display and storage

Program Development

The first task for many GPIB controllers is program development—writing, editing, and debugging the applications software that will control the system. This puts some special demands on the controller. For example, it should have a complete easy-to-use keyboard. In addition, user-definable keys, though not absolutely necessary, make menus or other input functions much simpler to implement and easier to use.

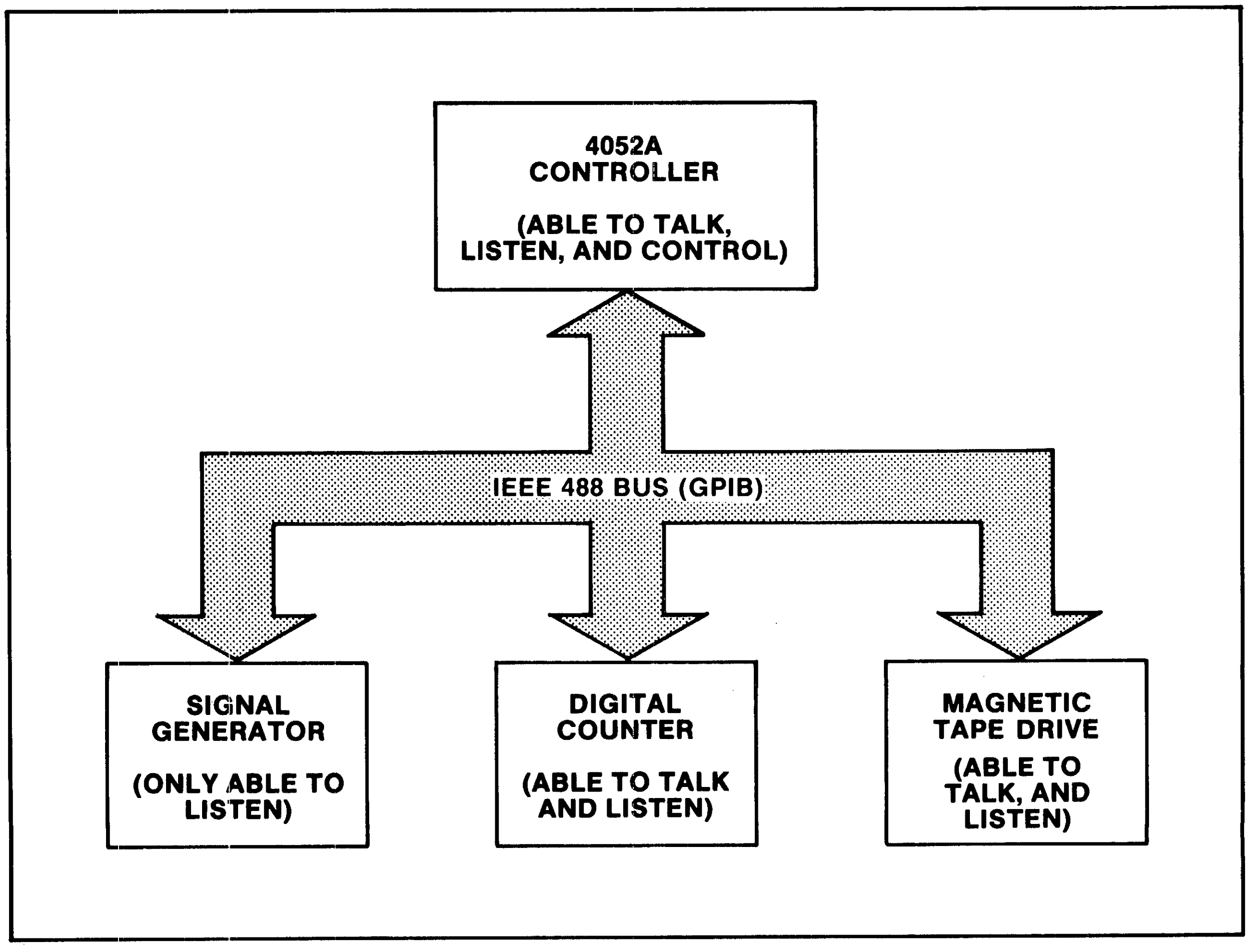


Fig. 1-1. *A typical GPIB system includes a controller and a variety of GPIB-interfaced devices with different capabilities.*

A full-size display is important for easy program listing and debugging. And, a convenient, powerful editor should be provided to create and modify program text. Finally, efficient mass storage is important for storing programs and data. A standard, transportable media allows programs developed on one controller to be transferred and run on many other systems.

Controlling the System

Next, consider the task of instrument control. No matter how powerful the system components are, if their actions are not coordinated, the system is like an orchestra without a conductor. The controller directs the entire system in performing its intended function. It assigns tasks to the instruments, coordinates communication, handles error conditions, and monitors the system's progress. The instrument control task can be further divided into five functions:

Addressing instruments
Sending commands
Transmitting and receiving data
Handling interrupts
Monitoring device status

Let's look at each of these functions individually:

Addressing Instruments. The controller selects an instrument or set of instruments to be involved in an operation by addressing them. Every instrument is assigned a unique primary address in the range 0-30. The controller uses this address to assign a device to talk or listen. In addition, some instruments have secondary addresses that select sub-sections or functions within the instrument. For example, Fig. 1-2 shows that the Tektronix 7612D Programmable

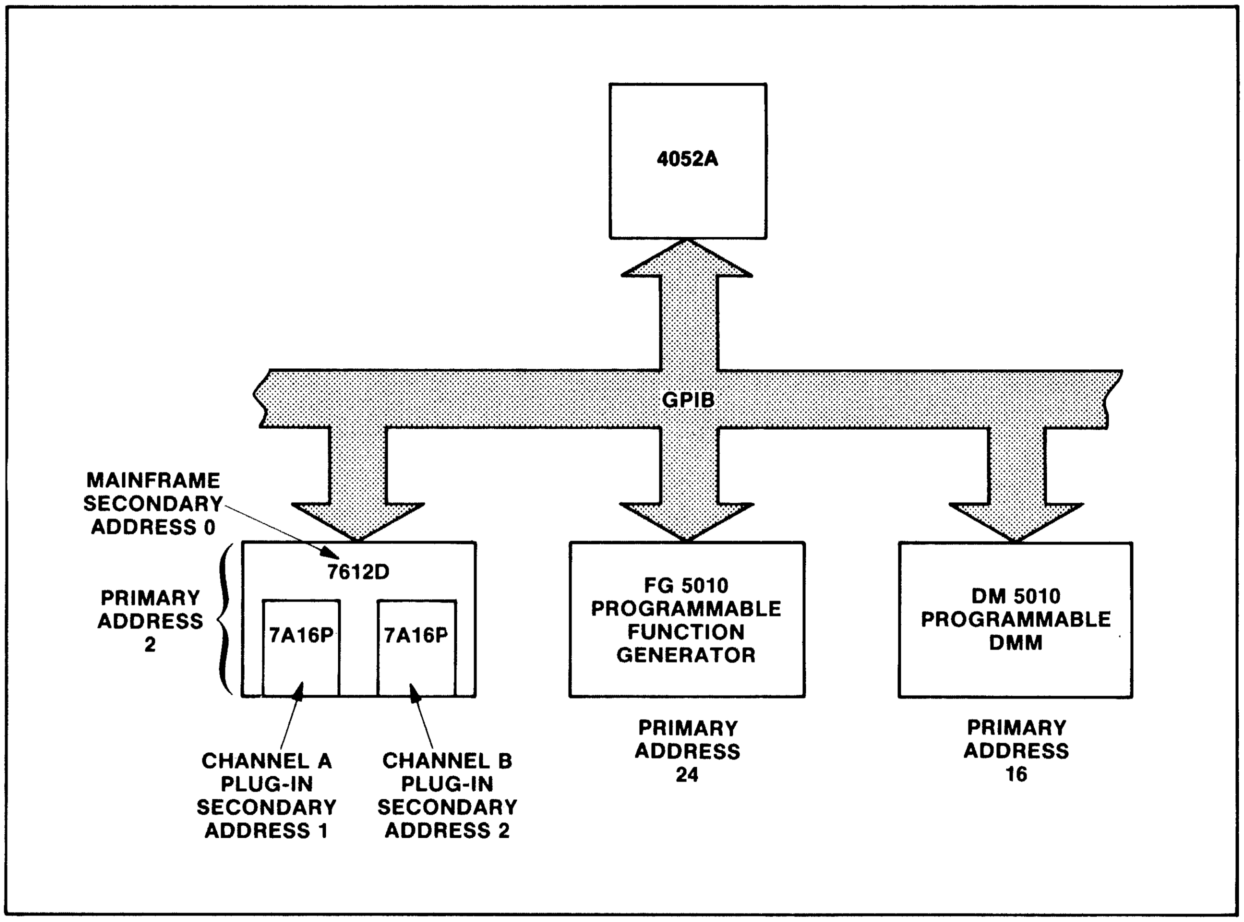

4052A
GPIB
MAINFRAME SECONDARY ADDRESS 0
PRIMARY ADDRESS 2
7612D
7A16P
7A16P
CHANNEL A PLUG-IN SECONDARY ADDRESS 1
CHANNEL B PLUG-IN SECONDARY ADDRESS 2
FG 5010 PROGRAMMABLE FUNCTION GENERATOR
PRIMARY ADDRESS 24
DM 5010 PROGRAMMABLE DMM
PRIMARY ADDRESS 16

Fig. 1-2. *Each instrument on the bus is assigned a unique primary address. Secondary addresses are used in some instruments to select sub-sections or functions within the instrument.*

Digitizer has a secondary address for its mainframe and one for each programmable plug-in.

Sending data and commands. The controller sends two basic types of messages: device-dependent messages and interface messages (Fig. 1-3).

Interface messages are commands that control interface functions. Interface messages come in two types: uniline and multiline. Multiline messages can be further subdivided into universal commands and addressed commands. Figure 1-4 shows how the different types of GPIB messages are related.

Multiline interface messages are sent by placing a byte on the GPIB with ATN asserted. Multiline interface messages may be either universal commands, affecting all devices on the bus, or addressed commands affecting only the addressed instruments, or addresses themselves.

Uniline interface messages are sent by asserting one line of the GPIB. Uniline interface messages

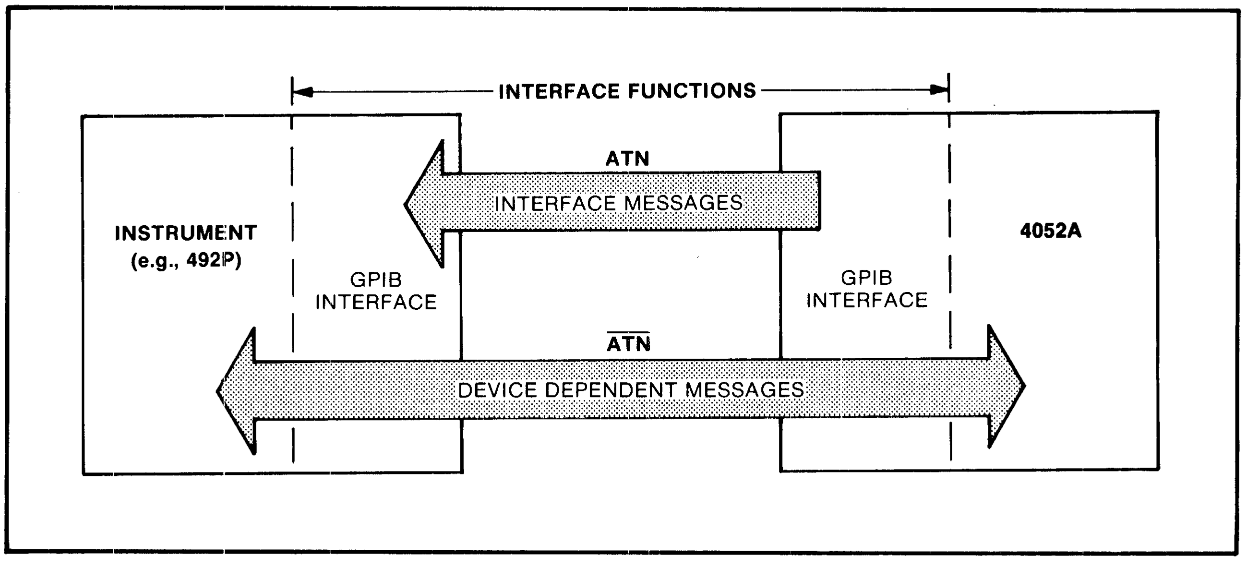


Fig. 1-3. *The controller sends interface messages with Attention (ATN) asserted. These messages control interface functions. Device dependent messages, sent with ATN unasserted, control instrument functions.*

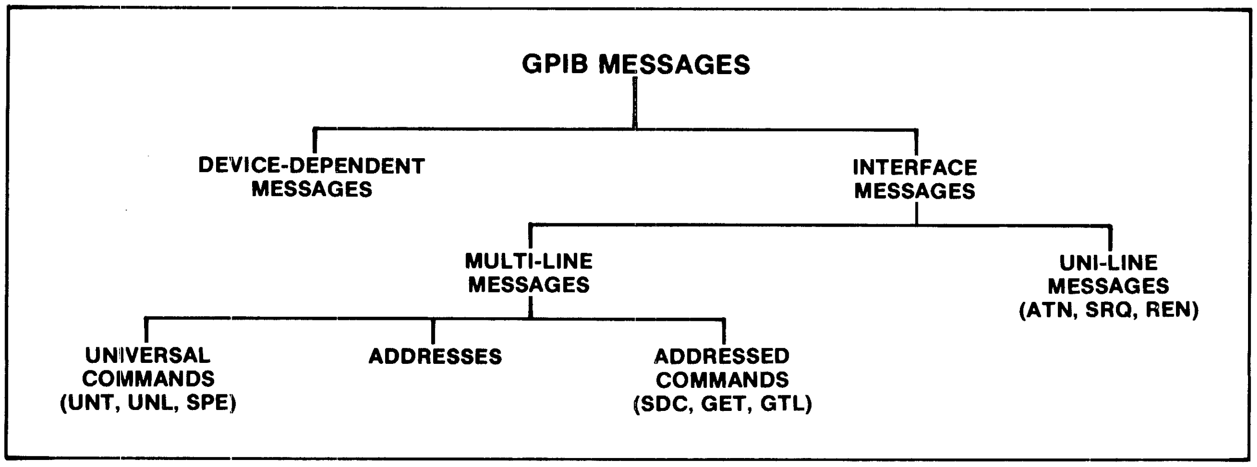

GPIB MESSAGES
DEVICE-DEPENDENT MESSAGES
INTERFACE MESSAGES
MULTI-LINE MESSAGES
UNI-LINE MESSAGES (ATN, SRQ, REN)
UNIVERSAL COMMANDS (UNT, UNL, SPE)
ADDRESSES
ADDRESSED COMMANDS (SDC, GET, GTL)

Fig. 1-4. *Messages sent over the GPIB can be divided into two general types—interface messages and device-dependent messages. Interface messages are further divided into universal multi-line messages, addressed multi-line messages, and uni-line messages.*

include ATN (attention), IFC (interface clear), SRQ (service request), REN (remote enable), and EOI (end or identify).

Device-dependent messages consist of commands or data that control instrument functions. The content and format of these messages is not specified in the IEEE 488 standard; it is left to the instrument designer. The messages may consist of queries that return instrument settings or data, commands that control instrument settings, or other data, such as waveforms. Device-dependent messages are always sent with the GPIB attention (ATN) line unasserted.

Transmitting and receiving data. Most instruments send data to and/or receive data from the system controller. A digitizer, for example, acquires waveform data and transmits it to the controller for processing and storage. A programmable spectrum analyzer, such as the Tektronix 492P might receive processed waveform data from the controller for display on its CRT. Data may be transmitted using a variety of codes including binary or ASCII.

Handling interrupts. Devices in the system generate interrupts to inform the controller of error conditions, the completion of an operation, or other asynchronous events that require the controller's attention. The controller finds the device that generated the interrupt (if there is more than one instrument in the system), reads its status, and takes appropriate action.

Processing the Data

The second major task of a GPIB system controller is processing the data acquired from instruments. Often, a few important parameters must be extracted from a mass of raw acquired data. Again, the system controller takes over. A few instruments, such as the 492P Programmable Spectrum Analyzer and the 7854 Programmable Oscilloscope, can do some processing internally. But many can only send raw data to the controller, depending on it for processing.

This processing may involve simple operations, such as signal averaging or finding the amplitude of an acquired pulse. More advanced applications may require operations such as the fast Fourier transform (FFT) or convolution. Powerful high-speed minicomputers have made lengthy and complex calculations feasible even in a small GPIB system controller, a task once left to large mainframe computers. With this power, the controller can set-up the instruments, acquire test data, and compute the desired parameters from the acquired data—all without human intervention.

Storing and Displaying the Data

Once data is acquired and processed, the controller is responsible for storing and/or displaying the results. Non-volatile mass storage, such as floppy disks or magnetic tape, provides a convenient means of logging data or results. In addition, the controller can generate graphic displays that make visual analysis of the data much easier. Hard copy (paper) output is also important for documentation of test results and displays. This output may be provided by a hard copy unit that copies the screen contents or by a plotter, or a printer.

GPIB Capability—
More than GPIB Compatibility

An efficient, powerful GPIB system requires more than just a computer with an IEEE 488 interface—it requires a **capable** controller with the hardware, software, and peripherals to handle the tasks. Many a frustrated user has found that an IEEE-488 interface and a plug on the rear panel do not make a good GPIB controller. There is a considerable difference between GPIB compatibility and GPIB capability!

The Tektronix 4050-Series Desktop Computers have long been known as the leader in high-performance graphics computers. But, they are also very capable GPIB controllers. The 4052A integrates a high-speed bit-slice minicomputer, a high-resolution graphics display (1024 x 780 points), an internal magnetic tape mass storage (up to 600 Kbytes), and a GPIB interface in a single compact unit (Fig. 1-5).

The 4052A's powerful hardware is supported by a greatly enhanced version of the popular, easy-to-learn 4050 BASIC originally implemented on the 4051. 4050 BASIC incorporates a flexible I/O structure that allows simple addressing of GPIB instruments and peripherals. It also includes extensions for signal processing, graphics, and

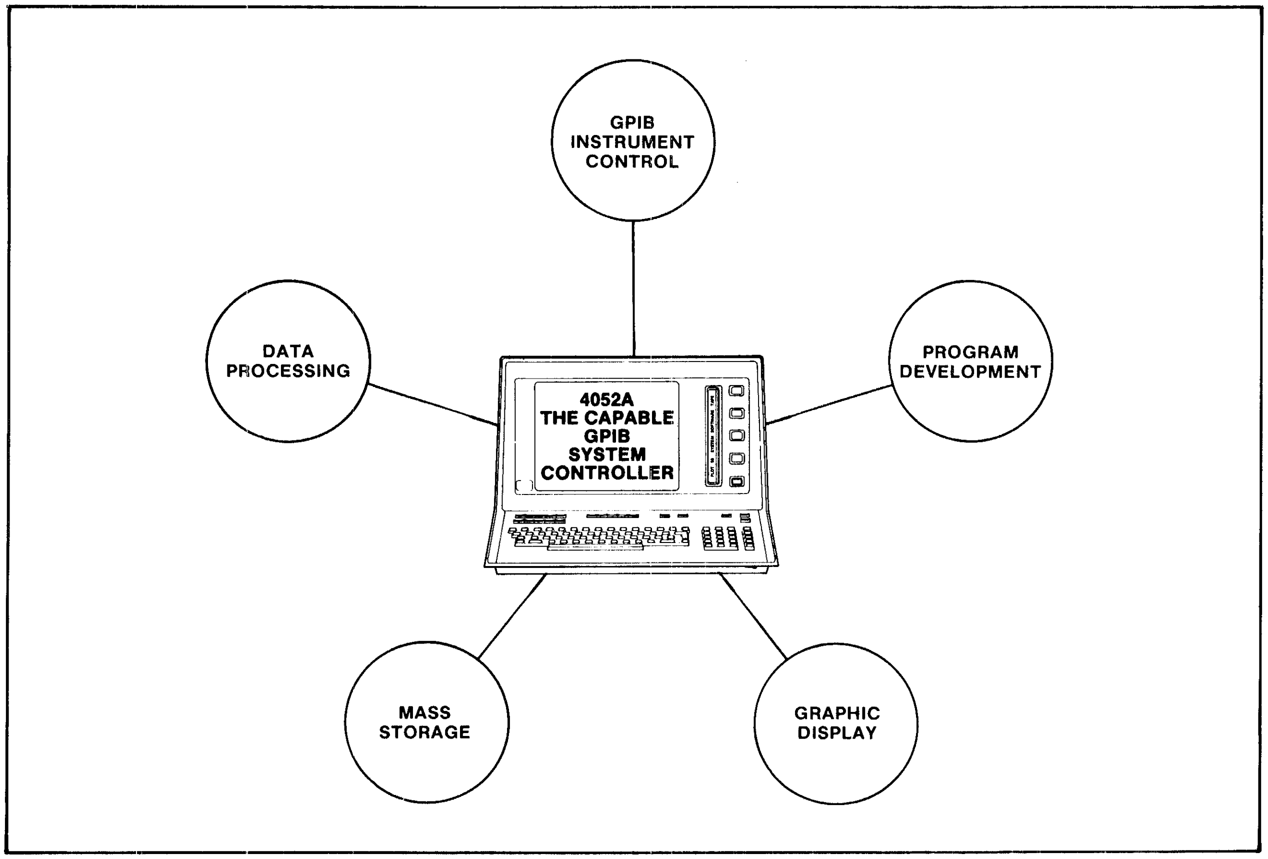


Fig. 1-5. *The 4052A is more than a GPIB-compatible desktop computer—it's a capable system controller.*

GPIB control. 4052A Basic has several features that facilitate programming and program maintenance:

- Program structuring
- Subprograms with parameters and local variables
- DO loops
- IF .. THEN .. ELSE constructs
- Multicharacter variable names

Thus, the 4052A houses all the essential ingredients of a powerful, flexible, and **capable** GPIB system controller in a single cabinet.

Section 2 — Configuring a 4052A GPIB System

The GPIB is a flexible interface—it can efficiently link many different types of instruments together to perform a variety of jobs. We have looked at the system controller's job and discussed some of the qualifications of a capable GPIB controller. But, choosing the right instruments and the right configuration for your system is also important. A clear definition of what you want the system to do and a basic understanding of the system components is the key.

Defining the System's Job

The first step in configuring a system is to define its job. Consider these questions:

1. **What is the system's operating environment?** Will it be performing repeated tests on a production line? If so, speed is probably a primary concern. On the other hand, accuracy is often more important in a research environment, where an operator sitting at a keyboard probably won't notice an extra second or two of delay.

2. **Will the system need to generate test stimuli?** If so, one or more signal sources will be required. And if the output of the source must be changed during a test, the generator(s) should be programmable.

3. **Will the system acquire data?** If the system is intended to make automated measurements, some type of data acquisition instrument is needed. The acquisition could be as simple as a DC voltage measurement, or as complex as a high-speed digitization of a transient waveform. The important points to consider here are the type of data to be acquired, number of data channels, and the GPIB capabilities required in the acquisition instrument. Also remember that the 4052A must be programmed to receive the acquired data. A variety of data formats are used, so be sure you know the specifics of how your instrument transmits its data. We'll talk more about this later.

4. **Will the data need to be processed?** If the acquired data requires processing, the controller or instrument must be capable of performing the necessary computations in the available time.

5. **Will data or test results be logged to a peripheral device?** In some cases, where data must be captured very quickly, data logging may be necessary. Data can be written to a peripheral device, sometimes without even passing through the 4052A. Later, when the acquisition is complete, the 4052A can read and process data from the peripheral at a slower rate. It may even be set up to log data from an acquisition, initiate the acquisition, and process the data from the last acquisition while the next one is in progress.

Getting on the Right Path

These are some of the questions that need answers as you begin configuring your GPIB system. It's not an exhaustive list, but answering these questions will get you on the path to a clear definition of your system's job. And that's a big step toward a well-designed, efficient system.

Selecting System Components

With a clear definition of the system's purpose in mind, you can begin selecting the specific instruments to accomplish that purpose. This discussion focuses on the GPIB considerations of selecting components. Other required specifications will be determined by the application.

Is the instrument really programmable? Often, instruments that are described as "IEEE 488 programmable" in catalogs and sales brochures are actually only partially programmable. Some functions can only be set from the front panel or by internal controls. It's important to know which functions, if any, are NOT programmable when you are selecting instruments. For each component in the system, you should have a list of the functions that must be programmable. If, for example, you need a function generator, your list might include programmable frequency, phase, and symmetry. As you look for programmable function generators, look at the specifications carefully. Are these functions programmable? Don't assume that the functions you need will be programmable just because the brochure says the instrument is "programmable".

In addition, most Tektronix programmable instruments provide a convenient query command that returns the current instrument settings to the controller. The settings are returned in a format that can be stored directly and transmitted back to the instrument as commands. Query commands are also included to return individual instrument settings or parameters.

This feature is especially valuable in interactive systems where the operator may make manual adjustments to the instrument through the front-panel controls. The operator may set-up the controls manually and send the settings to the controller by pressing a front-panel button or issuing a command from the controller.

How fast is the instrument? Speed can be an important factor in choosing GPIB system components, particularly for systems that are intended to perform high-speed tests in a production environment. The speed of a GPIB instrument is determined by three basic factors: the time required for acquisition, internal processing, and data transfer. If your system will be performing in an environment where speed is critical, take a careful look at the data transfer rate and other speed specifications of the instruments. Section 6 describes some techniques for estimating the performance of a GPIB system.

What interface functions are implemented? A device's GPIB interface provides the link between the GPIB and the programmable device functions. The IEEE 488 standard allows a designer to choose from a list of optional functions when implementing the device interface. These interface functions are defined in terms of the following "interface subsets":

Source Handshake
Acceptor Handshake
Talker
Listener
Service Request
Remote/Local
Parallel Poll
Device Clear
Device Trigger
Controller

The instrument designer can choose to implement all, part, or none of each of these functions, as defined by the function subsets in the standard. You should find a list of the interface subsets in the specifications for any GPIB instrument. The list may sound strange until you realize that it's just a shorthand way of describing the device's interface functions. C0, for instance, says that an instrument has no capability as a controller. DT1 means that an instrument can be triggered to perform a designer-chosen function when it receives the group execute trigger interface message. A summary of the interface subsets is contained in appendix A.

As you select instruments for the system, keep these interface subsets in mind, but don't confuse them with the programmable functions of a device. The interface subsets only describe the capabilities of the device's GPIB interface, not the programmable functions of the device itself.

All instruments in a system do not need to have the same interface subsets. But, the capabilities of some instruments may not be useable unless other instruments in the system, or the controller also implement the same interface subsets.

Consider, for example, the Device Trigger (DT) interface subset. Instruments that have the device trigger function implemented (DT1 interface subset) can be set up to start acquiring data or initiate some other process when they receive the Group Execute Trigger (GET) interface message. This function is useful when several instruments must be synchronized to perform a test. However, if only one of the instruments in the system has the DT1 interface subset, the device trigger feature won't be very useful, since the other instruments in the system don't understand the GET message and can't be triggered by it (Fig. 2-1).

Addressing. The IEEE 488 standard defines the basic addressing scheme for GPIB instruments. However, it leaves several options open to the instrument designer, so it's also important to know the individual addressing requirements of the instruments you are considering.

All GPIB instruments have at least one primary address in the range 0-30. In most cases, this is the only address the user must be concerned with. The instrument actually has one address for talking (if it can talk) and one address for listening (if it can listen). But the 4052A automatically generates these "absolute" talk and listen addresses from the single primary address (except for the low-level WBYTE and RBYTE statements). When you use an output statement like PRINT, the controller adds 32 to the primary address to generate the absolute listen address of the instrument. When you use an INPUT statement, the controller adds 64 to the primary address to generate the absolute talk address. In most cases, this process is automatic, so the user need only remember the single primary address.

Some instruments also have one or more secondary addresses. This address selects a sub-

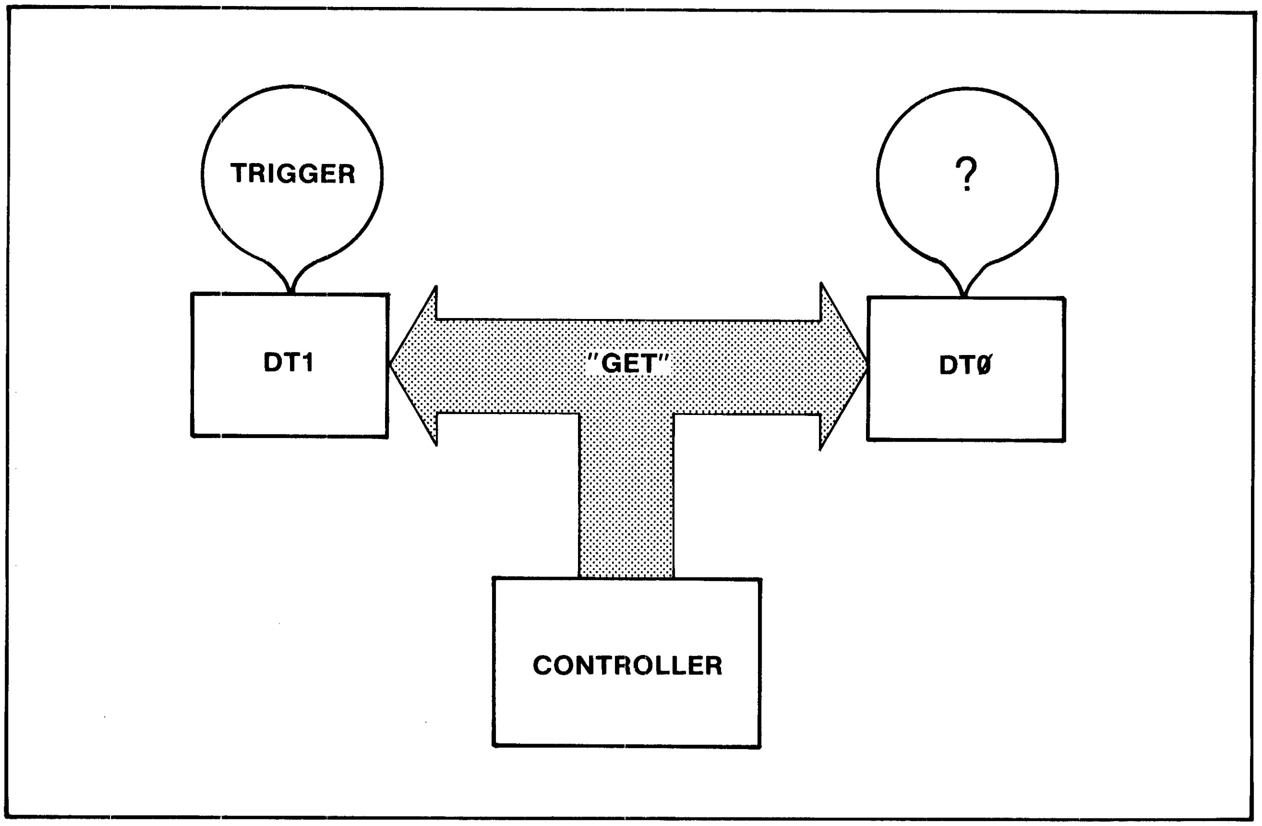


Fig. 2-1. *An example system using Group Execute Trigger (GET) to initiate an operation. Both the function generator and the digitizer must implement the DT1 interface subset (Device Trigger capability).*

function or part of the instrument to take part in the operation. The specific use of this secondary address is not defined in the standard, so manufacturers use it in several different ways. Again, the user specifies an address in the range 0-30, and the controller automatically adds 96 to this address to generate the absolute secondary address.

Primary GPIB addresses are usually set by a set of five switches inside the instrument or on the rear panel. These switches allow you to set the address from 0 to 31, but there are some limitations. Some controllers, such as the 4052A, reserve address zero for themselves, so you can't use address zero with these controllers. Also, 31 is not a valid address—it is used for the universal UNTalk and UNListen commands. Setting a device to address 31 effectively eliminates it from the bus since it can never be addressed. A typical set of address switches is shown in Fig. 2-2.

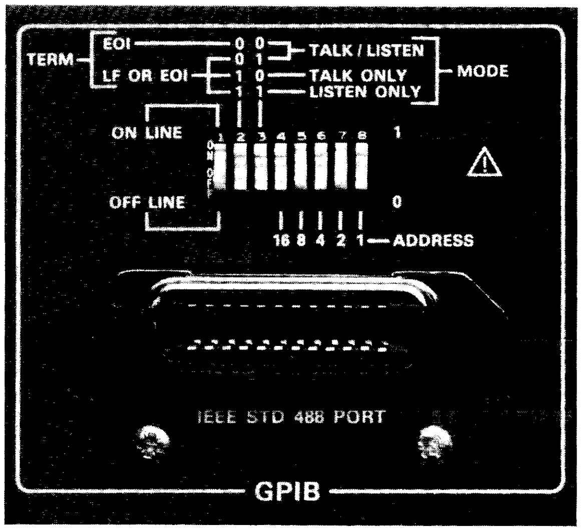


Fig. 2-2. *7854 Oscilloscope's GPIB connector and selection switches for setting primary address and communication mode.*

If a secondary address is required, it is usually set by a separate switch. In the 7912AD and 7612D, the secondary address switch sets the mainframe secondary address. The secondary address of the left plug-in is the mainframe secondary address plus one. The right plug-in address is the mainframe address plus two. So, to address the left plug-in to listen, the primary listen address is sent, followed by the secondary address of the left plug-in. We'll look at some specific examples of how this is accomplished in 4050 BASIC later.

Talkers and listeners. Instruments in the system can take one of three roles: Talker, Listener, or Controller. Since the 4052A will not allow any other device to take control of the bus, we'll only consider talkers and listeners.

At this point, it's important to understand what the IEEE 488 standard refers to as a "talk-only" or "listen-only" instrument. These terms refer to instruments that can be manually configured (usually with a switch) as permanent talkers or permanent listeners. When configured in this mode, the instruments do not need to be addressed by a controller. They are permanently addressed and they participate in every bus transaction. Other instruments may only be capable of talking or listening, but if they must be addressed by a controller, they are not considered "talk-only" or "listen-only" devices as defined by the standard.

Talk-only and listen-only instruments are useful when a small system is set-up without a controller. Often, the system simply consists of a talk-only acquisition instrument, such as the 468 Digital Storage Oscilloscope, and a peripheral configured for listen-only operation, such as the 4924 Digital Cartridge Tape Drive (Fig. 2-3). In this configuration, the acquisition instrument sends its data to the tape drive for logging. No other bus traffic occurs and a controller is unnecessary.

When talk-only or listen-only instruments are not used, the controller assigns the role of talker or listener to an instrument by issuing its talk or listen address, respectively. Unaddressed instruments do not participate in the transaction.

When choosing system components, it's important to know which instruments need to talk, which ones need to listen, and which ones need to do both.

Message format. Another important consideration when you are configuring a GPIB system is the message format used by each instrument. The syntax and coding of device-dependent messages is not specified in the IEEE 488 standard. As a result, there is no universal standard for message coding. This can be a source of frustration when programming the system, because of the widely different message formats used.

Tektronix has developed a codes and formats standard designed to enhance compatibility among its GPIB instruments. The standard specifies message coding and syntax designed to be unambiguous, correspond to those used by similar devices, and be as simple and obvious as possible. This standard makes programming a system of

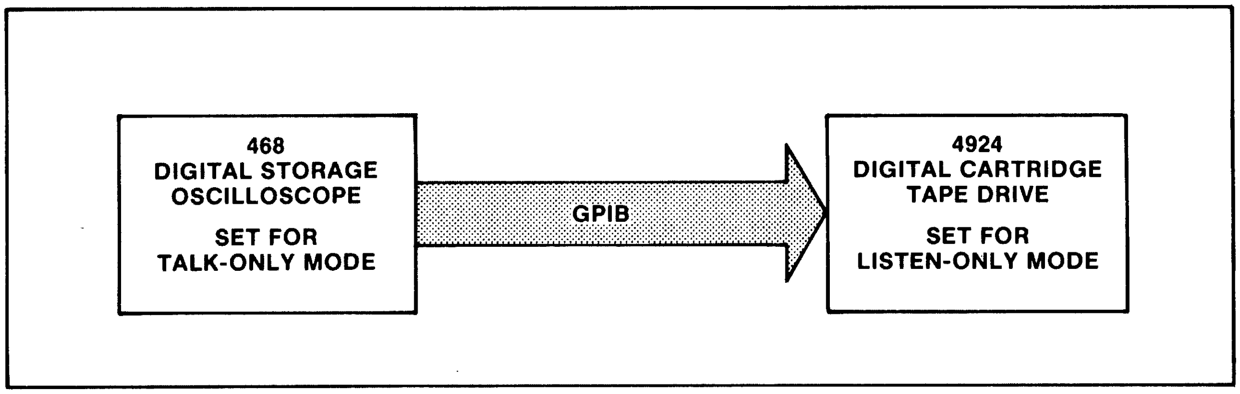


Fig. 2-3. *Some instruments can be manually set to permanent talker (talk-only mode) or permanent listener (listen-only mode). This allows small systems, such as the 468/4924 system shown here, to operate without a controller. These instruments may also be operated with a system controller.*

Tektronix GPIB instruments easier and simpler, because the messages for all instruments are similar and easy to remember. And since the commands consist of simple English-like mnemonics, programs are easier to read and understand.

Message terminators. Manufacturers also use different techniques to indicate the end of a message. Some instruments assert the EOI bus line when they are finished talking, others send a special character, such as line-feed. Again, the key is knowing what the instruments require. Using Tektronix instruments eliminates most of these problems, since they are designed to conform to the codes and formats standard, which specifies EOI as the message terminator. Most Tektronix instruments can also be configured to use the line-feed terminator when operated with other controllers.

Getting It Together

Now that you understand the capabilities and requirements for each instrument in your system, the job of actually configuring the system should be simple. The following paragraphs provide a few guidelines for connecting the instruments together and setting the bus addresses.

Setting the bus address. The first step is setting the bus addresses for each instrument. Remember that every device must have a unique address. Valid primary addresses are 0-31, but don't use address zero—the 4052A reserves this address for itself. Also, selecting address 31 logically removes the device from the bus; it does not respond to any address, but remains both unlistened and untalked.

If you change the address switches after an instrument is powered-up, the address may not actually be updated until you return to local, re-initialize, or turn power off and back on. Check the instrument manuals for more details.

Since the 4052A POLL command allows you to sequentially poll instruments in any order, it is not necessary to arrange the addresses according to interrupt handling priority. As you set the addresses, write each one down for reference when writing programs.

Setting the message terminator. The message terminator on most instruments is selected with a switch on the rear panel or an internal strap. The most common delimiters are line feed and EOI. Tektronix controllers use EOI, but line feed option is available for compatibility with other controllers.

Cabling the instruments. The next step is cabling the instruments together. Up to 15 devices, connected by not more than 20 meters total cable length, can be interfaced to a single IEEE 488 bus. In some cases, more than 15 devices can be interfaced if they do not connect directly to the bus, but are interfaced through another device. For example, this scheme is used for programmable plug-ins housed in a 7612D or 7912AD Programmable Digitizer.

The system can be cabled in a star or linear configuration (Fig. 2-4). To maintain the bus electrical characteristics, a device load must be connected for each two meters of cable. Although devices are usually spaced no more than two meters apart, they can be separated farther if the required number of device loads are lumped at any point. If a single instrument is interfaced to a controller, the two-meters-per-instrument rule allows the controller and instrument to be separated by four meters of cable.

Generally, at least two-thirds of the instruments on the bus should be powered-up for correct operation. In some cases, the bus will operate properly with fewer instruments powered-up. Check the standard for more details.

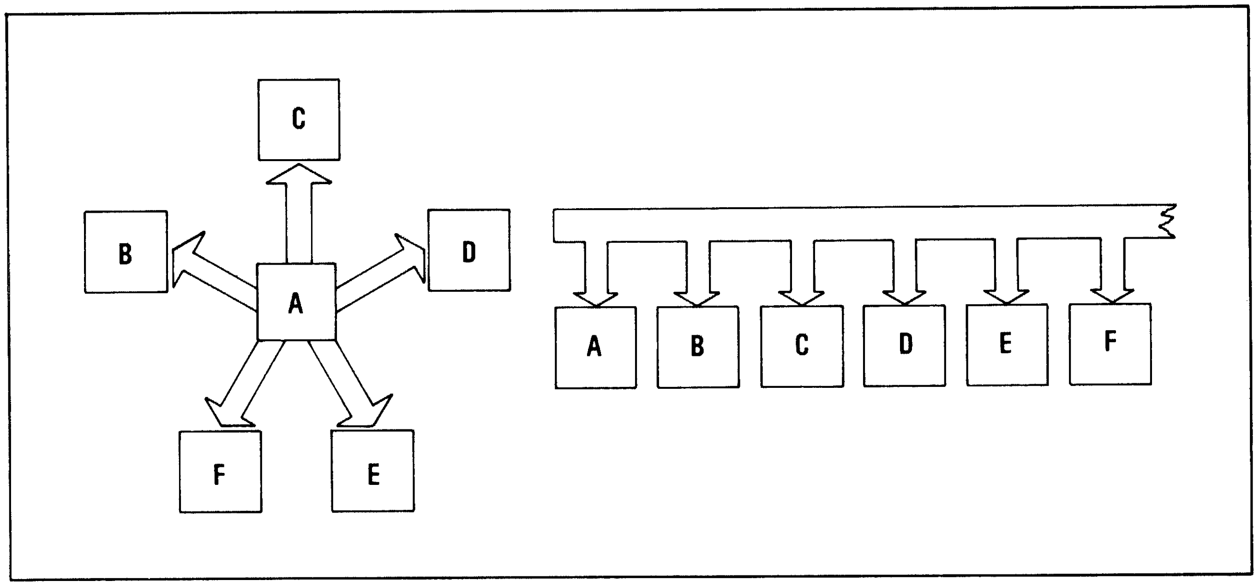


Fig. 2-4. *The GPIB system can be configured in either a star or linear manner.*

Section 3 — Programming the 4052A

Introduction to 4050 BASIC

The 4052A runs an enhanced version of 4050 BASIC. 4050 BASIC contains extensions in graphics, file system access, I/O operations, matrix operations, character string manipulation, high-level language interrupt handling, and operating system facilities. The 4052A implementation adds the following capabilities:

- GPIB enhancements
- Program structuring
- Multi-character variable names
- Binary operations
- Additional graphics commands
- Additional array handling

Although these extensions provide considerably more power than standard BASIC, most of the extensions are exercised through optional entries in the statements. This enhances compatibility with most other BASIC languages.

Additional language extensions in GPIB operations, signal processing, real-time control, enhanced graphics, and file editing are available using optional ROM packs.

The 4052R14 Option 1A GPIB Enhancement ROM pack eases the GPIB programmer's job by offering routines that do binary transfers, error trapping, parallel polling and several other operations.

If your controller will have to process waveforms or other array data, the signal processing ROM packs (4052R07 and 4052R08) will be particularly valuable. These two ROM packs provide 15 waveform and array processing functions, including differentiation, integration, maximum, minimum, and cross functions, fast Fourier transform, inverse Fourier transform, convolution, correlation, and others.

In process control and other real-time applications, the 4052R09 Real Time Clock ROM Pack is also very useful. It provides five time and date functions with elapsed time measurement and a programmable interrupt.

All ROM pack routines are accessed with a simple CALL statement. More information on the ROM packs is provided in Section 5—"Processing and Displaying Data."

I/O Addressing in 4050 BASIC

4050 BASIC uses a powerful I/O addressing technique that handles all peripherals—internal and external—the same. For example, the same PRINT statement can be used to write data on the internal magnetic tape or to send ASCII data to an external GPIB-interfaced device. The only difference is the address specified in the statement. Let's look at a typical PRINT statement to see how this addressing technique works:

```
PRINT @33,12:"THIS IS ASCII DATA"
```

The keyword PRINT tells BASIC that data is to be output. The next two numbers are primary and secondary addresses. If they are not specified, the PRINT statement causes the data to be printed on the graphic display screen. When the addresses are included in the statement, the data is sent to the specified address. Addresses may be specified as constants, variables, or numeric expressions.

Each peripheral device in the system is assigned a primary address. For example, the above PRINT statement sends the ASCII string "THIS IS ASCII DATA" to the device at address 33, the internal magnetic tape drive. Primary addresses are assigned as shown in Table 3-1.

TABLE 3-1
DEVICE NUMBER ASSIGNMENTS

Device Number	Device
1-30	External devices on the GPIB
31	4052A Keyboard
32	4052A graphic display
33	4052A magnetic tape drive
34	DATA statement
35-36	Unassigned
37	Processor status
38-40	4050E01 ROM Expander
41	Left-most ROM slot
42-50	4050E01 ROM Expander
51	2nd-from-left ROM slot
52-60	4050E01 ROM Expander
61	3rd-from-left ROM slot
62-70	4050E01 ROM Expander
71	4th-from-left ROM slot
72-80	Unassigned

Each I/O statement in 4050 BASIC has a default primary address that refers to the internal peripheral usually accessed with that keyword. However, any valid primary address may be substituted for the default value. Table 3-2 lists the default addresses for each I/O statement.

TABLE 3-2
DEFAULT I/O ADDRESSES

BASIC Statement	Default I/O Address
APPEND	@33,4:
CLOSE	@33,2:
COPY	@32,10:
DRAW	@32,20:
FIND	@33,27:
FONT	@32,18:
GIN	@32,24:
HOME	@32,23:
INPUT	@31,13:
KILL	@33,7:
LIST	@32,19:
MARK	@33,28:
MOVE	@32,21:
OLD	@33,4:
PAGE	@32,22:
PRINT	@32,12:
RDRAW	@32,20:
READ	@34,14:
RMOVE	@32,21:
SAVE	@33,1:
SECRET	@37,29:
TLIST	@32,19:
WRITE	@33,15:

The second number in the BASIC statement is a secondary address. Internal 4052A peripherals and some external peripherals use the secondary address to determine what type of I/O action is required. For example, if a KILL statement is executed, the internal tape drive is addressed by default. The default secondary address of 7 tells the tape drive that a KILL operation is being executed. The same operation could be executed using a PRINT statement by specifying the primary address of the tape drive, and the secondary address for the KILL operation. Thus, these two statements are equivalent:

```
PRINT @33,7:n = KILL n
```

where n is the number of the file to be KILLed.

If a secondary address is specified in an I/O statement, it is sent in place of the default address. If you specify 32 as the secondary address, no secondary address is sent.

Some instruments, like the Tektronix 7912AD and 7612D Programmable Digitizers share a single primary address among the mainframe and up to two programmable plug-ins installed in the mainframe. These instruments use the secondary address to select the mainframe or one of the plug-ins for involvement in an I/O operation. Others, like the TM 5000 series of programmable instruments, use a separate primary address for each instrument. Since the Tektronix TM 5000 mainframe has no programmable functions, it is not assigned an address. When addressing instruments that use the secondary address for selecting a sub-function, simply specify the correct primary and secondary address for all I/O to the device.

4050 BASIC also simplifies addressing of external GPIB devices. A primary address in the range of 1 to 30 is used to address a device, whether to talk or listen. When an instrument is addressed to listen, 4050 BASIC adds 32 to the primary address to generate an absolute listen address. When an instrument is addressed to talk, 64 is added to the primary address. For example, if an instrument's primary address is set to 1, it listens at absolute address 33 and talks at absolute address 65. The 4052A automatically generates these absolute addresses from the primary address specified in the I/O statement.

The secondary address is also specified as a number in the range 0-30. 4050 BASIC adds 96 to this value to generate the absolute secondary address.

I/O Statements

The 4052A's I/O statements can be separated into three levels as illustrated in Fig. 3-1. The highest level of statements perform special I/O functions that make programming easier and more efficient. For example, the DRAW statement makes graphics much simpler and faster than implementing the same function with PRINT statements.

The next level of I/O statements are designed for simple operations such as sending or receiving ASCII data. The PRINT and INPUT statements provide some data formatting for ASCII input or output. The READ and WRITE statements perform a similar operation for machine-dependent binary input and output.

The lowest level of I/O statements are the RBYTE and WBYTE statements. These statements are intended to provide line-level control of the GPIB

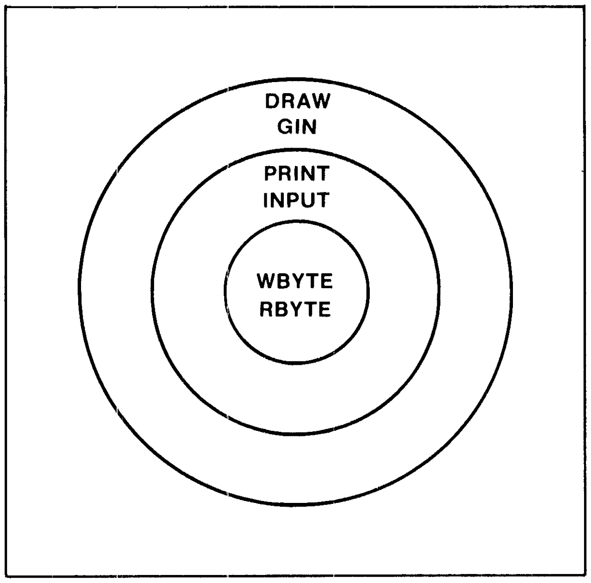


Fig. 3-1. *4052A I/O statements can be divided into three levels, from high-level statements like DRAW and GIN that implement special graphics I/O functions, to low-level statements like WBYTE and RBYTE that provide line-level control of the GPIB data bus.*

data bus at the expense of speed and increased complexity. RBYTE and WBYTE are exceptions to the standard addressing rules in 4050 BASIC. The absolute talk or listen address and the absolute secondary address (if required) must be specified, instead of the primary addresses used in higher level statements.

Bytes are transferred in straight binary code. The programmer is responsible for checksums or other error checks. These statements give you full control of the GPIB data bus, and the ATN (Attention) and EOI (End or Identify) lines. The statements also allow you to set up a transfer between two instruments, without passing the data through the 4052A.

GPIB data transfers are discussed in more detail in Section 4.

Interrupt Handling

4050 BASIC also provides a simple high-level facility for handling service request (SRQ) and other interrupts. Statements are included to perform serial polls, transfer program control asynchronously on an interrupt condition, wait for an interrupt condition, or disable interrupts. Interrupt handling capabilities of 4050 BASIC are discussed in more detail in Section 4.

Program Structuring in 4052A BASIC

4052A BASIC has features that enable you to write structured programs. These features include:

- Subprograms with local variables and parameter passing
- IF ... THEN ... ELSE constructs
- DO loops

4052A BASIC also allows you to use multicharacter identifiers and comment tails, thus improving readability.

Binary Operations

4052A BASIC allows you to perform the following binary operations:

- AND
- OR
- exclusive OR
- complement
- rotate
- shift
- test
- set

These binary operations are performed bit-by-bit on string quantities. If you want to perform binary operations on numeric quantities, you must first translate the numeric into an appropriate string. Figure 3-2 is a subprogram that AND's two numerics and stores the result in a numeric.

```
500 SUB _AND(IN1,IN2,_RESULT)
510      LOCAL IN1$,IN2$,OUT$
520      IN1$=STR(IN1)
530      IN2$=STR(IN2)
540      CALL "BITAND",IN1$,IN2$,OUT$
550      _RESULT=VAL(OUT$)
560 END SUB
```

Fig. 3-2. *This program AND's two numerics and stores the result in a numeric.*

Section 4 — Programming a 4052A GPIB System

Writing the programs that control a GPIB system is often the most time consuming and difficult part of building the system. But, with a clear definition of the system's purpose, carefully chosen components, and a powerful programming language like 4050 BASIC, the job is greatly simplified.

This section provides a guide for writing 4050 BASIC programs for a 4052A-controlled GPIB system. The details of reading and writing commands and data, interrupt handling, and interface control are covered. A generous supply of sample programs are included. We'll also take a brief look at GPIB peripherals, such as floppy disk drives, tape drives, and plotters.

System Power Up

Power up test. When it's time to power your system up, there are a few things you'll need to be ready for. First, remember that most programmable instruments automatically perform some kind of self-test procedure on power up. The instruments usually won't respond to any front-panel or GPIB input until the power up test is complete—all you can do is wait. The time required to complete this procedure varies from milliseconds to several seconds.

If all goes well in the test, the instrument powers up normally. Otherwise, errors are usually reported on the front panel and by setting the status byte to indicate the error. When the self-test is complete or errors are detected, the instrument asserts the GPIB SRQ line to tell the controller that its status can be read.

Power up SRQ. In its initialized state (after power up or INIT), SRQ's are disabled (ignored) on the 4052A. To enable SRQ's, your program must execute an ON SRQ THEN statement. The following statement enables SRQ's and specifies statement 2000 as the beginning of the SRQ handling routine.

```
100 ON SRQ THEN 2000
```

After executing this statement, the 4052A responds to SRQ's by doing an implicit GOSUB to line 2000.

Since the SRQ line is shared among all instruments on the bus, the 4052A can't tell which instrument(s) are asserting it. Nor can it tell, at this point, whether the instruments completed their power up tests normally, or if errors were detected. The solution is to read the status byte from each instrument. This accomplishes two things: First, it tells the 4052A if the instruments powered up normally, and if they didn't, what's wrong. This information can be passed on to the system operator via the controller's display. Second, reading the status byte clears the SRQ.

Poll statement. Use the POLL command to read the status byte from each instrument. The command format is:

```
POLL Device,Status;<primary address,[secondary address]>
```

(The secondary address is optional—use only when required.)

In the simplest case, where a single instrument is connected to the 4052A and set for bus address 1, the command is:

```
POLL Device,Status;1
```

The instrument's status byte is returned in the variable STATUS and a 1 is returned in DEVICE, indicating that the first device polled was asserting SRQ. Several addresses may be specified in a single POLL command by separating each address with a semicolon. Secondary addresses may also be included by separating them from the primary address with a comma. For example:

```
POLL Device,Status;1,1;2;3
```

This statement first polls the device at address 1 with secondary address 1. This might be a programmable plug-in installed in a programmable mainframe. If this device is not requesting service, the next device in the list (address 2) is polled. If this one isn't asserting SRQ, the last instrument (address 3) is polled.

When the instrument that is asserting SRQ is found, its position in the list of addresses is returned in the first variable (DEVICE in our example). The status byte from this instrument is returned in the second variable (STATUS). The polling stops with the first instrument that is found asserting SRQ. If several instruments are asserting SRQ, the POLL statement must be executed once for each device asserting SRQ. With an ON SRQ statement and an appropriate address list in the POLL statement, the repeated polling will happen automatically.

Configure routine. Instead of listing each address you want to poll in the POLL statement, you can put the addresses in an array and just specify the array name in the POLL statement. You don't have to load the array element-by-element; the CONFIGURE routine will do it for you.

To illustrate, assume that a system contains the following instruments set for the addresses shown:

Instrument	**Primary Address**	**Secondary Address**
7612D Digitizer	3	0
7A16P Amplifier plug-in	3	1
7A16P Amplifier plug-in	3	2
DC 5010 Counter/Timer	20	-
FG 5010 Function Generator	22	-

When the system is powered up, all five instruments assert SRQ. The program shown in Fig. 4-1 polls the instruments, and prints the status bytes on the 4052A screen.

```
100 DIM A_list(30,2)
110 CALL "CONFIGURE",E,A_list
120 ON SRQ THEN 2000
1980 WAIT
1990 GOTO 1980
2000 DO    ! SRQ HANDLING ROUTINE
2010   POLL Device,Status;A_list
2020 EXIT IF Device=0
2030   PRINT "SRQ FROM DEVICE AT ADDRESS ";
2040   PRINT A_list(Device,1);". STATUS ";Status
2050 LOOP
2060 RETURN ! END OF SRQ HANDLING ROUTINE
```

Fig. 4-1. *A program to poll instruments and print the status bytes on the 4052A screen.*

Line 100 dimensions the address list array (A_LIST) to 30 rows (the maximum number of primary addresses) and two columns (column 1 for primary addresses, column 2 for secondary addresses).

Line 110 calls the CONFIGURE routine to fill A_LIST with the addresses of all the devices on the GPIB that have a listen address. After the CONFIGURE routine is executed, A_list looks like this:

3	0
3	1
3	2
20	-1
22	-1

A "-1" in the second column indicates there is no secondary address. When you use an address list array with the POLL statement, the "-1" is properly interpreted.

Line 120 enables SRQ's and specifies line 2000 as the beginning of the SRQ handling routine.

Line 1980 causes the 4052A to wait for an interrupt.

Line 1990 jumps to the WAIT statement after the SRQ handling routine returns.

Lines 2000 to 2050 contain a DO loop that repeatedly polls the devices on the GPIB until none is asserting SRQ.

Line 2010 polls the devices whose addresses are in A_LIST. When it polls a device that is asserting SRQ (the instrument sets bit 7 in the status byte when it is asserting SRQ), the polling routine stops and places the status byte in STATUS and the device position (the corresponding index into A_LIST) in DEVICE. If no device is asserting SRQ, DEVICE is set to zero.

Line 2020 exits the DO loop when no more devices are asserting SRQ.

Lines 2030 and 2040 print a message containing the GPIB primary address and the status byte of a device that was asserting SRQ.

Line 2060 returns control to the program line following the line that was executing when the SRQ occurred.

SRQ interrupts can also occur for a number of reasons other than power up. We'll discuss interrupt handling in more detail later in this section.

NOTE

To locate a device on the GPIB, the CONFIGURE routine sends a listen address and waits for a device to assert NDAC. If NDAC is not asserted after a prescribed amount of time (1 ms default), the CONFIGURE routine goes on to the next address. Thus, the CONFIGURE routine will not locate a device that can not be listen addressed.

Device Dependent Messages

Device dependent messages compose the vocabulary of a GPIB instrument. The content and format of these messages is not defined by the IEEE 488 standard; it is determined by the instrument designer to suit the needs of the particular instrument. Device dependent messages may include queries that return instrument settings or acquired data, commands that control instrument settings, or data such as waveforms or measurement results.

This subsection describes the format of device-dependent messages for Tektronix instruments as well as the techniques for transmitting and receiving these messages with the 4052A.

Device dependent message I/O. Regardless of the message content, coding, or format, the basic process of transferring a device-dependent message is the same. The message is always transferred from a device addressed as a talker to one or more devices addressed as a listener. The process is illustrated in Fig. 4-2.

First, the talker and listener(s) must be assigned. The controller asserts the ATN line and puts the appropriate address on the bus. If a secondary address is required, it directly follows the primary address. If, for example, the 4052A is sending a message to a device, the listen address of that device is placed on the bus. The 4052A automatically assumes the role of talker when outputting a message and that of a listener when receiving a message.

When the addressing sequence is complete, the 4052A releases attention and puts the first byte of the message on the bus. The bytes are transferred one at a time at the rate of the slowest listener until all bytes are sent. Then, the 4052A asserts ATN again and sends the universal UNTalk and UNListen commands to clear the bus.

Fortunately, most of the mechanics of transferring messages is transparent to the user; the 4052A takes care of it. In the special cases, low-level commands are provided that allow you to control bus activity more directly.

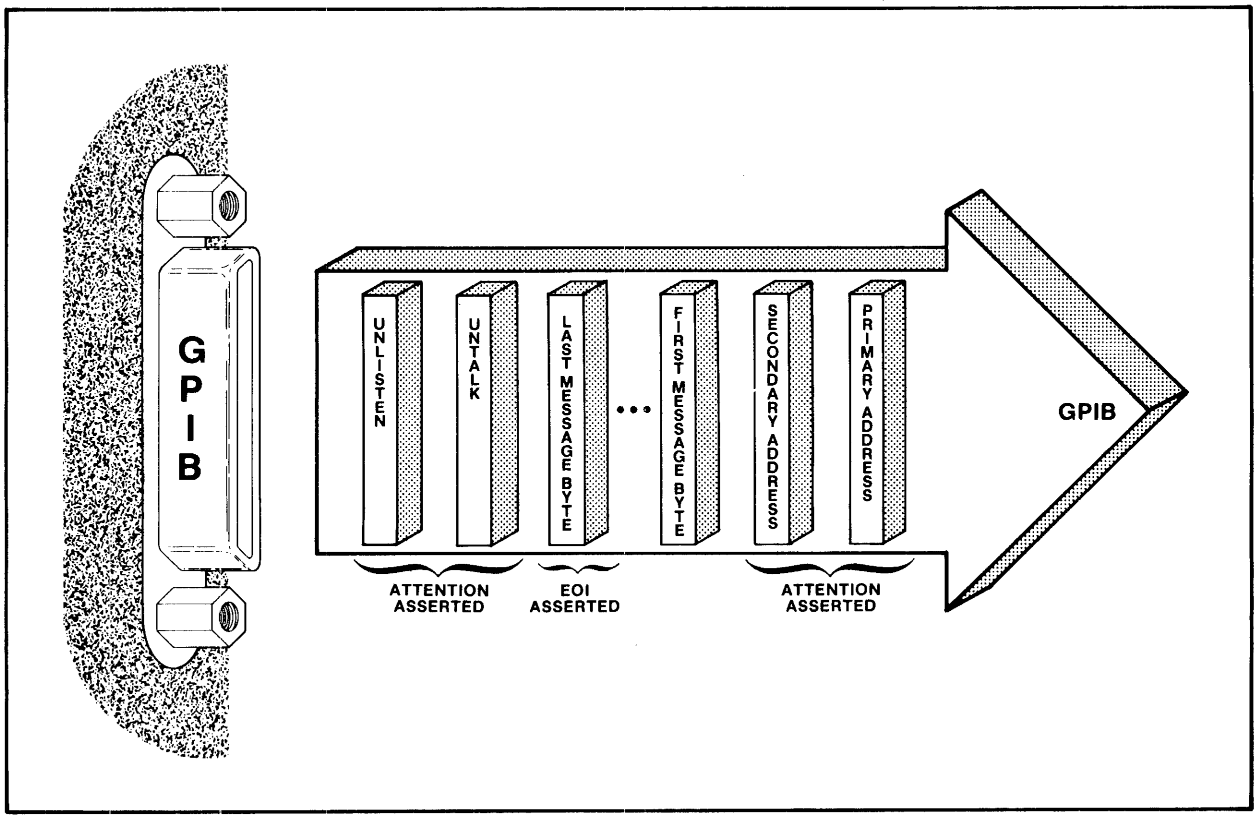


Fig. 4-2. *The basic process of transferring device-dependent messages is the same, regardless of the content and format. A primary address is sent first, followed by a secondary address (if required). Then, the data bytes are sent, followed by the UNTalk and UNListen commands.*

Set commands. Device dependent messages that set instrument operating modes or parameters are called set commands. In all Tektronix GPIB instruments, these commands take the following form:

```
<header>[<space><arguments>][<semicolon>]
```

The header is a mnemonic or keyword that identifies the command. If the command requires arguments, they are separated from the header by a space. The arguments specify the values or parameters required by the command. For example, the TIME AUTO command in the 492P Programmable Spectrum Analyzer sets the time/division to automatic mode. TIME is the command header and AUTO is an argument. Using the MAN argument (TIME MAN), sets the time/division to manual mode. Other commands, such as FREQ 1MHZ (set center frequency to one megahertz), require numeric arguments. Still others require no arguments at all, such as SIGSWP (single sweep).

Multiple set commands can be grouped together and sent as a single message by separating the commands with semicolons. For example:

```
TIME AUTO;FREQ 1MHZ
```

combines the TIME and FREQ commands into a single message.

Since these commands are composed of ASCII characters, a 4052A PRINT statement can be used to send the command string to the instrument. Assume, for instance, that a 492P is connected to the bus at address 10. The commands shown above could be sent using the statement:

```
PRINT @10:"TIME AUTO;FREQ 1MHZ"
```

Alternately, the command string could be stored in a string variable, say COMMAND$, and transmitted by specifying the variable name in the PRINT statement as shown below.

```
Command$="TIME AUTO;FREQ 1MHZ"
  :
PRINT @10:Command$
```

Expanding this technique allows you to build command strings dynamically within a program. In the program below, the 4052A prompts the operator to enter the desired center frequency for the 492P. The operator's response is used to build the command string.

```
10 PRINT "ENTER THE DESIRED CENTER FREQUENCY :";
20 INPUT C_freq$
30 PRINT @10:"TIME AUTO;FREQ ";C_freq$
```

Line 10 prompts the user for the center frequency and line 20 stores the response in the variable C_FREQ$. Then, line 30 appends C_FREQ$ to the end of the command string and sends the completed string to the instrument. Expanding this concept with a few simple statements would allow the program to check the response for validity, report errors to the operator, and request valid input.

A numeric variable could also be used in the above example in place of the string variable, C_FREQ$. When a numeric variable is used in the PRINT statement, the value is automatically converted to a string of ASCII digits before it is transmitted. The original variable is unaffected. Numeric variables can also be used with INPUT statements. The conversion from ASCII input to internal numeric format is automatic.

Query commands. Query commands are device-dependent messages that cause the instrument to return information about its settings or operation. The form of the query commands is:

```
<header><question mark>[<semicolon>]
```

Many query commands are simply set command headers with a question mark added. For example, the FREQ command for the 492P Programmable Spectrum Analyzer becomes a query simply by changing it to FREQ?. The FREQ? query returns the current center frequency setting.

Query commands can be sent in a PRINT statement just like set commands. However, since the instrument returns a message in response to the command, the 4052A must also accept the response. Most query responses are sent in ASCII, so an INPUT statement can be used to receive it. The format of the query response is:

```
<header><space><response>[<semicolon>]
```

where response is the setting, value, or function returned by the query.

For example, to get the current center frequency setting from an FG 5010 Programmable Function Generator, send the FREQ? command. When the FG

5010 receives this command, it expects to be addressed to talk so it can send the reply. The following program segment illustrates this process.

```
10 PRINT @24:"FREQ 2E6"
20 PRINT @24:"FREQ?"
30 INPUT @24:C_freq$
```

Line 10 sets the output frequency to 2 megahertz. Then, line 20 sends the FREQ? query. Line 30 gets the complete reply (header and value) and puts it in the string variable C_FREQ$. In this example, C_FREQ$ would contain the string:

```
FREQ +2.0E+6;
```

If you only want the numeric value from the response, simply specify a numeric variable in the INPUT statement, instead of a string variable. The header (FREQ) will be ignored and the value will be returned in the specified variable. In the previous example, if a numeric variable had been specified, say F0, the FREQ header would be ignored and the value of the current frequency returned in F0.

Set and query commands may be grouped together in a single message by separating the commands with semicolons. For example, the two PRINT statements in the previous example could be condensed into a single statement by combining the set and query commands as follows:

```
10 PRINT @24:"FREQ 2.0E+6;FREQ?"
```

In most instruments several queries may be included in a single message. However, the specific rules for multiple commands in a message vary slightly among instruments, so refer to the Operator's or Programmer's Manuals for your instruments.

Sending ASCII data. The controller may send waveform or other data to instruments for processing or display. The format of this data depends on the instrument, but many accept it in ASCII-coded decimal numbers. For example, the 492P can accept ASCII waveform data using the CURVE command. Several options are available with this command, but for the sake of example, consider its simplest form:

```
CURVE <data value><comma or space><data value>
<comma or space>...
```

A typical CURVE command might be:

```
CURVE 27,28,29,31,33,36,39,44,...
```

The 4052A can transmit the CURVE command and ASCII data using a simple PRINT statement such as

```
PRINT @1:"CURVE ";Curve_data;
```

where CURVE_DATA is an array. When this statement is executed, the 4052A addresses the 492P and sends the CURVE command header. Then, it begins sending the contents of array CURVE_DATA. Transmission starts with the first element, CURVE_DATA(1), and ends with the last. The semicolon on the end of the statement suppresses the extra spaces added by the 4052A between array elements. It is not required, but it reduces the number of bytes sent by eliminating the extra spaces and, thus, speeds up the transfer.

Array elements may not be sent using separate PRINT statements because each statement asserts EOI with the last byte transmitted, so each value is sent as a separate message. The instrument requires that all data points be sent in a single message.

The data can also be transmitted from a string array by substituting the string variable name for the numeric variable and deleting the trailing semicolon, as shown below.

```
PRINT @1:"CURVE ";A$
```

When the data is transmitted from a string variable, the contents of the string are transmitted without modification. Processing the data is more difficult when it is stored in a string, since individual data elements cannot be easily accessed. But, transmitting data from ASCII strings is faster than transmitting from a numeric variable because the 4052A does not have to perform any data conversion. For simple waveform storage, when no processing is required, storage in string variables is probably best. When processing is required, the data should be stored in numeric variables.

Reading ASCII data. Data can also be received from an instrument in ASCII-coded decimal numbers. The 4052A INPUT statement accepts data from the instrument and stores it either in numeric variables or string variables. For example, the statement

```
INPUT @10:A
```

reads a single ASCII number and stores it in the variable A. If A was previously dimensioned as an array, the 4052A attempts to read one number for each element in the array (or until the input device asserts EOI). When more than one number is received with a single INPUT statement, individual numbers must be delimited by a non-numeric character (valid numeric characters are 0-9, +, -, and in scientific notation, E). Most instruments delimit each data value with a space or comma.

Also, since non-numeric characters are ignored in numeric input, this can be a handy way to strip off unwanted data headers and store only the numbers. For example, when the 492P sends a waveform in response to a CURVE? query, a waveform header precedes the data. A typical CURVE? query response is shown below.

```
CURVE CRVID:FULL,27,28,29,31,...
```

If a numeric array is specified in the INPUT statement, the 4052A ignores the ASCII header characters and begins storing data with the first waveform value (27). Each value is stored in successive array elements.

ASCII data can also be read into a string variable by specifying a string variable in the INPUT statement. When data is read into a string, all ASCII characters are stored exactly as sent. Reading data into a string is faster than reading into numeric variables because no data sorting or conversion is necessary. And, since query responses can be directly transmitted back to the instrument as a command, string storage is an efficient means of saving instrument parameters, waveforms, or settings for sending back to the instrument later.

Using alternate delimiters on INPUT. When multiple variables or an array is received with a single INPUT statement, the individual elements must be properly delimited. For numeric input to multiple variables or arrays, the delimiter is simple. Any non-numeric character (characters other than 0-9, +, -, and E in scientific notation) is a valid delimiter.

INPUT to string variables isn't always so simple. The problem is that the only valid delimiters for string input are EOI, carriage return, or a designated alternate delimiter.

String input can be broken into parts using the alternate input delimiter feature in the 4052A. If a percent sign (%) is specified in place of the at sign (@) in the I/O address for the INPUT statement, the 4052A uses a previously specified ASCII character for a delimiter. The delimiter character is defined by modifying the processor status parameters with a PRINT statement.

```
PRINT @37,0:N1,N2,N3
```

The first number (N1) specifies the ASCII decimal code for the alternate INPUT delimiter. It must be in the range of 0-255. If, for example, 65 is specified, the ASCII letter "A" delimits string INPUT just as non-numeric characters delimit numeric input. But remember, the alternate delimiter is only used when the INPUT %N form is used.

The second number (N2) specifies the ASCII decimal code for the end-of-file character. This value must be in the range 0-255, and it must be specified, whether or not it is changed from the default (255).

The third number (N3) specifies the ASCII decimal code for the character that will be deleted from incoming ASCII strings. If, for example, 67 is specified, all upper case C's are deleted from the input. This value must be specified whether or not the default (255) is changed.

Once the alternate delimiter is specified, an INPUT %N: statement can be used to read data up to the delimiter character. If, for example, you want to read ASCII CURVE data from a 492P and store the waveform identifier separately from the data, the routine shown in Fig. 4-3 will do the job.

```
10 REM *** SPECIFY ALTERNATE DELIMITER ***
20 PRINT @37,0:44,255,255
30 REM *** NOW READ THE DATA ***
40 PRINT @10:"CURVE?"
50 INPUT %10:W$,A
```

Fig. 4-3. *A program to read ASCII data from the 492P using alternate delimiters.*

Line 20 changes the INPUT delimiter to a comma (ASCII code 44). Then, line 40 sends the CURVE? query and line 50 begins by reading the curve identifier into W$. The identifier is separated from the first data value by a comma, so data storage in W$ stops at the end of the header. The remaining data values are stored in array A.

Binary waveform data format. The 4052A can also transmit waveform data in binary as required by some instruments. For example, the 492P can accept waveform data in binary as well as ASCII. Binary data transmission is slightly more complex than ASCII transmission, but it's considerably faster.

The Tektronix codes and formats standard specifies two formats for binary data transmission. Some instruments, like the 492P, can accept data in either format. Others, like the 7912AD Programmable Digitizer, require one format. Check your instrument Operators or Programming manual for the required format.

The first format is called the "end block binary" format. It is simple, but has no provision for error checking. The format is:

```
@<data value><data value>...
```

@ is the ASCII code for the "@" character. This tells the instrument that binary data in the end block binary format follows.

DATA VALUE is an 8-bit binary number. If the instrument requires 16-bit values, two bytes are sent for each value, most significant byte first. EOI is asserted with the last byte in the data block.

The second format, called the block binary format, is more complex since it includes a byte count and checksum for error checking. This format is:

```
%<byte count><data value><data value>...<checksum>
```

% is the ASCII code for a "%" character. This tells the instrument that a binary block with error checking follows.

BYTE COUNT is a 16-bit binary number that indicates the number of bytes remaining in the message, including the checksum. The byte count is sent as two bytes, most significant byte first.

DATA VALUE is an 8-bit binary number. If the instrument requires 16-bit values, two bytes are sent for each value, most significant byte first.

CHECKSUM is an eight-bit binary number that is the two's complement of the modulo-256 sum of all preceding bytes except the first (%). That is, the two's complement of the eight-bit sum of the preceding bytes, ignoring the carry. If the receiver computes a modulo-256 sum of all the bytes except the percent sign, but including the checksum, the result should be zero. Thus, the checksum provides an error check for the binary block transmission.

Sending binary data. Binary data can't be transmitted with the PRINT statement because the 4052A converts all data in a PRINT statement to ASCII before sending it. However, 4050 BASIC provides a low-level I/O statement that gives the programmer line-level control of the GPIB: WBYTE (Write Byte). WBYTE can send any byte on the GPIB with or without attention (ATN) and/or EOI asserted. In its general form, the syntax of the WBYTE statement is:

```
WBYTE @<attention byte>...:<device-dependent bytes>
```

When WBYTE is used to send device-dependent messages, the syntax is:

```
WBYTE @<listen address>[,<secondary address>...]:bytes>...
```

All bytes sent before the colon are sent with attention asserted; all bytes after the colon are sent with attention unasserted. Any byte in the range +255 through -255 may be sent. The magnitude (0-255) determines the byte sent; the sign determines the state of the EOI line. Positive values are sent with EOI not asserted; negative values cause the byte to be sent with EOI asserted.

The addresses specified in the WBYTE statement must be absolute physical addresses, instead of the peripheral device number. The listen address for an instrument is simply its peripheral device number plus 32, the talk address is the peripheral device number plus 64, and the secondary address is the peripheral device number plus 96. For example, to address an instrument set for primary address 1 and secondary address 1, use the statement:

```
WBYTE @33,97:<data bytes>...
```

This statement tells the 4052A to assert attention and send the listen address (32+1=33). Then, with attention still asserted, it sends the secondary address (96+1=97). After releasing attention, the device-dependent data bytes following the colon are sent.

As an example, the following program segment addresses the 492P to listen and sends binary data stored in array A.

```
10 WBYTE @33:ASC("@"),A,-255
20 WBYTE @63:
```

Line 10 of the program first sends the 492P listen address (33) with attention asserted. Then, with attention unasserted, the ASCII code for "@" is sent, followed by the data array (A). The -255 byte is sent as a message terminator. The 492P receives this byte, but it is not included in the waveform data. The last byte of the waveform data could also be negated and used as a message terminator, eliminating the need for the -255 byte. Line 20 uses the general form of WBYTE to send an interface message, UNListen (63) with attention asserted. This message tells all addressed listeners to stop listening.

If an ASCII command header, such as CURVE or LOAD, is required, the header is sent before the @ or % characters.

Generating the byte count for block binary. The block binary format includes a byte count for error detection. The byte count indicates the number of bytes remaining in the message, including the checksum.

To compute the byte count, simply add 1 to the size of the data array to account for the checksum.

The byte count is a 16-bit quantity; it must be divided into two bytes before being transmitted (most significant byte first). If the byte count is smaller than 255, set the high-order byte to zero. Dividing the byte count into two bytes requires only two lines of 4050 BASIC. The following program segment takes a byte count value stored in BYTECOUNT and converts it to an equivalent two-byte value in BYTECOUNT_HI and BYTECOUNT_LO.

```
10 Bytecount_hi=INT(Bytecount/256)
20 Bytecount_lo=Bytecount MOD 256
```

To see how this works, consider an example. A byte count of 515 is represented in binary as:

1000000011

Since this number is expressed in ten bits, it must be divided into two bytes. Line 10 of the program divides the byte count by 256, truncates the result to an integer, and stores it in BYTECOUNT_HI. In this case, BYTECOUNT_HI contains a 2. Since the value of the least significant bit in BYTECOUNT_HI is 256, the 2 in BYTECOUNT_HI actually represents a value of 512.

Line 20 of the program finds the smallest number that is congruent to BYTECOUNT, modulo 256, and stores it in BYTECOUNT_LO. Thus, the byte count is represented in two bytes as:

BYTECOUNT_HI	BYTECOUNT_LO
00000010	00000011

With the byte count separated into two bytes, it can be transmitted with a WBYTE statement.

Generating the checksum. The block binary format also includes a checksum byte for error checking. The checksum is computed by taking the modulo-256 sum of all bytes in the block except the % character and the checksum. The result is converted to its two's complement before transmission. When the instrument receives the bytes, it computes the modulo-256 sum of the bytes and adds the checksum. If the result is zero, the transmission is assumed to be correct.

The following statement computes a checksum for binary data stored in array A.

```
10 Checksum=256-(SUM(A)+Bytecount_hi+Bytecount_lo MOD 256)
```

Subtracting from 256 converts a byte value to its two's complement.

Sending block binary data. With the byte count and checksum computed, the complete binary block can be sent with two simple statements. The program in Fig. 4-4 shows the process of computing the byte count and checksum, transmitting the binary block, and unaddressing the instrument.

```
10 Bytecount=UBOUND(A,-1)+1
20 Bytecount_hi=INT(Bytecount/256)
30 Bytecount_lo=Bytecount MOD 256
40 Checksum=256-(SUM(A)+Bytecount_hi+Bytecount_lo MOD 256)
50 WBYTE @34:ASC ("%"),Bytecount_hi,Bytecount_lo,A,-Checksum
60 WBYTE @63:
```

Fig. 4-4. *A few simple 4050 BASIC statements calculate the byte count and checksum, and transmit the binary block.*

Line 10 uses the UBOUND function to determine the number of elements in array A. Lines 20 and 30 split BYTECOUNT into two bytes and line 40 computes the checksum. Then, line 50 transmits the binary block. The block begins with the ASCII code for "%", followed by the byte count in BYTECOUNT_HI and BYTECOUNT_LO. Next comes the data array (A) and the checksum (CHECKSUM). Negating CHECKSUM causes the 4052A to assert EOI when it is sent. Line 60 sends the UNListen message to unaddress the instrument.

Reading binary data. Many instruments send waveform or other data to the controller in binary. Tektronix instruments send binary data in either the block binary or end block binary format previously described. In either case, the RBYTE statement in 4050 BASIC is used to read the data.

The syntax of the RBYTE statement is:

```
RBYTE <numeric variable>,[<numeric variable>]...
```

RBYTE simply accepts bytes from the talker and assigns them to the variables in the list. If an array variable is specified in the list, the 4052A reads data from the bus and begins filling the array starting with the first element. It continues to read data into the array until it is full. If EOI is asserted with a byte, the value is negated before it is stored in the variable.

Before RBYTE can receive data from an instrument, the instrument must be told what to say. This is usually accomplished with a PRINT statement (e.g., PRINT @1:"CURVE?"). Then, it must be addressed to talk. When receiving ASCII data with the INPUT statement, this process is automatic. When receiving binary data with RBYTE, the talk address must be manually sent using the WBYTE statement.

The program in Fig. 4-5 illustrates this process by reading a waveform from the 492P.

Line 10 changes an internal status flag in the 4052A. It tells the 4052A to delimit INPUT strings with the character whose ASCII code is 37 (%). Then, line 20 sends a message to the 492P that tells it to transmit data in binary and requests the data. Since the ASCII waveform header comes first, the INPUT statement in line 30 reads this header. The INPUT operation stops when the alternate delimiter selected by line 10 (%) is reached.

The first character in the binary block is a "%", so the INPUT operation stops reading at that point. Line 40 addresses the 492P to talk again, and line 50 gets the waveform data. The first two bytes are the byte count. They are stored in BYTECOUNT_HI and BYTECOUNT_LO. The data is read into the previously dimensioned array A. The last byte is a checksum, which is stored in CHECKSUM. Line 60 sends the UNTalk message to unaddress the 492P. Line 70 adds the bytes that were received by the RBYTE statement. The sum should be congruent to zero, modulo 256. If not, a data transmission error has occurred and line 80 prints an error message.

Sending Interface Messages

Sometimes it may be necessary to send an interface message or sequence of messages that are not implemented in a high-level 4050 BASIC statement. With the 4052A, you can send any GPIB interface message except SRQ.

Multiline messages are sent with the WBYTE statement. The following statement sends the DCL (Device Clear) interface message.

```
WBYTE @20:
```

```
10 PRINT @37,0:37,255,255
20 PRINT @1:"WFMPRE ENC:BIN;CURVE?"
30 INPUT %1:Header$
40 WBYTE @65:
50 RBYTE Bytecount_hi,Bytecount_lo,A,Checksum
60 WBYTE @95:
70 IF Checksum+Sum(A)+Bytecount_hi+Bytecount_lo MOD 256 = 0 THEN
80   PRINT "CHECKSUM ERROR"
90   STOP
100 ENDIF
```

Fig. 4-5. *A program to read binary data from the 492P using alternate delimiters.*

Since the byte (decimal 20) is before the colon, attention is asserted and the byte is interpreted as an interface message. Any multiline interface message can be sent by substituting the proper byte code(s) for the 20 in the statement above.

Uniline messages are sent in various ways depending on which one you are sending.

- ATN - You can assert ATN with any byte you put on the GPIB by placing it before the colon in the WBYTE statement.
- IFC - You can send the Interface Clear message with the CALL "IFC" statement.
- REN - You can assert REN with the CALL "RENON" statement. To unassert REN, use the CALL "RENOFF" statement.
- EOI - You can assert EOI with any byte you put on the GPIB with the WBYTE statement by negating the byte (X becomes -X).

Transfers Among GPIB Devices

The 4052A can also set up a transfer between two or more devices on the bus without being involved in the transfer. For example, you might want to transfer data from a tape drive to a plotter directly. If the data is stored in a format that the plotter understands, the 4052A does not need to be involved and the transfer may be faster without its involvement.

The WBYTE statement allows you to assign a talker and one or more listeners and then relinquish control of the bus to the talker, waiting for the EOI line to indicate that the transfer is complete. The following example illustrates how this is accomplished.

```
150   ON EOI THEN 180
160   WBYTE %70,109,52,35:
170   WAIT
180   WBYTE @63,95:
```

When line 150 is executed, the 4052A is told to transfer control to line 180 when EOI is asserted (the ON statement will be described more fully later in this section). Line 160 sends primary talk address 70 followed by secondary address 109. The primary address assigns this device as a talker. In this case, the secondary address tells the peripheral that it should transmit ASCII data in the upcoming transfer. Next, primary listen address 52 and primary listen address 35 are sent to assign these devices as listeners. The percent sign (%) in the WBYTE statement tells the 4052A to get off the bus and let the assigned talker take over when the ATN line is released.

At this point, the talker takes over the bus and starts sending data to the two listeners. Meantime, the 4052A waits at line 170 for the transfer to complete. The talker must assert EOI with the last byte of the message to signal the 4052A that the transfer is complete. When EOI is asserted, program control is transferred to line 180 and the 4052A assumes control of the bus again. It asserts ATN and sends the UNListen (63) and UNTalk (95) messages.

When specifying primary and secondary addresses in a WBYTE statement, only one device may be assigned as a talker, but up to 14 devices may be assigned as listeners (there can only be 15 devices on the bus). Secondary addresses and interface messages may also be sent in the message as necessary.

Interrupts and Instrument Status

Interrupt conditions. Sometimes special conditions, called interrupts, occur that cause the 4052A to temporarily suspend normal program flow. You can specify routines to handle these interrupts.

The five interrupt conditions are:

- SRQ (Service Request) from an external GPIB device.
- EOI (End Or Identify) from an external GPIB device.
- EOF (End Of File) from the internal magnetic tape unit.
- SIZE errors caused by numeric underflow or overflow in a program.
- TIMEOUT caused by a GPIB transfer that takes longer than a specified time period.

Enabling and disabling interrupts. All interrupts are disabled upon power up and INIT. Being "disabled" doesn't mean the same thing for all interrupt conditions.

With SRQ and TIMEOUT, "disabled" means the 4052A ignores SRQ's and TIMEOUT's. No error occurs if an instrument asserts SRQ or an I/O transfer times out.

With SIZE and EOF, "disabled" means there is no link between the interrupt and an interrupt handler. If one of these interrupts occur when it is disabled, an error is processed and program execution is terminated.

With EOI, "disabled" means the same as it does for SIZE and EOF except that when the 4052A is involved in a GPIB transfer (as opposed to the situation where the 4052A only sets up the transfer between other devices), EOI interrupts are always enabled and handled internally.

The starting line number of each interrupt service routine is specified with an ON statement. This statement tells the 4052A where to transfer control when a specified interrupt occurs. One ON statement must be included for each interrupt service routine. The syntax of the ON statement is:

```
        ( EOF (0)  )
        | SIZE     |
ON     <  EOI       >   THEN line number
        | SRQ      |
        ( TIMEOUT  )
```

When an ON statement is executed, the 4052A establishes a link between the specified interrupt condition and program line number. Nothing else happens when the statement is executed. But when the interrupt condition occurs, the 4052A finishes executing the current statement, and then does an implicit GOSUB to the previously specified interrupt handling routine.

For example, a program might contain the statement

ON SRQ THEN 120

This statement tells the 4052A to GOSUB to line 120 when an SRQ occurs. When it is executed, nothing obvious happens, but a link is established in the 4052A between the SRQ condition and line 120. This link remains valid until another ON SRQ statement is executed, the 4052A power is turned off, an INIT statement is executed, or an OFF SRQ statement is executed.

To disable an interrupt, use the OFF statement. OFF provides a convenient means of disabling interrupt conditions set up with a previous ON statement. The syntax of the OFF statement is:

```
        ( EOF(0)   )
        | SIZE     |
OFF    <  EOI       >
        | SRQ      |
        ( TIMEOUT  )
```

EOF Interrupts. The EOF interrupt occurs when the internal magnetic tape unit reaches the logical end of a file. When the EOF condition is specified in an ON statement, the logical unit number (0) must be specified along with the keyword EOF. For example, the statement

ON EOF(0) THEN 500

transfers program control to line 500 when the logical end of the current magnetic tape file is reached. This facility can be used to read data from a file of unknown length or to find the end of an existing file to append data.

EOI Interrupts. An EOI interrupt is generated whenever an external device asserts the EOI line on the GPIB. In most cases, EOI is used to indicate the last byte of a message. The talker asserts this line with the last byte in the message.

This interrupt is normally only used when a transfer is set up between two devices without the 4052A's involvement. The 4052A relinquishes control of the bus to the talker for the duration of the transfer. Since EOI is asserted with the last byte in the message, it can be used to tell the 4052A when the external transfer is complete and it can take control of the bus again. The use of the EOI interrupt is described more fully in **Transfers among GPIB Instruments**, earlier in this section.

The ON EOI statement has no effect when the 4052A is involved (talking or listening) in the data transfer. The EOI is handled internally.

SIZE Interrupts. A SIZE interrupt is generated when a numeric overflow occurs in the current program. In general, SIZE errors are caused by computations that result in out-of-range numbers. The numeric range of the 4052A is -1.0E+308 to 1.0E+308.

TIMEOUT Interrupts. A TIMEOUT interrupt occurs when a GPIB I/O operation takes longer than a previously specified time period. The default timeout period is infinite at power up and INIT. Use the CALL "TIMSET" statement to change the timeout period.

SRQ Interrupts. Part of the GPIB system controller's responsibility is to handle SRQ interrupts from instruments on the bus. An instrument may assert SRQ for any number of reasons, including power up, command errors, internal errors, operation complete, etc. In any case, the instrument expects the controller to respond by reading its status byte. Reading the status byte accomplishes two things: it tells the controller why the instrument asserted SRQ, and it clears the interrupt.

The status byte contains information about the instrument's internal operations, error conditions, or other information that is important to the controller. The IEEE 488 standard reserves bit 7 of the status byte to indicate whether an instrument is asserting SRQ or not. If the instrument is asserting SRQ, it sets bit 7; if not, it clears bit 7.

Since the SRQ line is shared among all instruments on the bus, any one asserting SRQ causes the line to be asserted. As a result, the 4052A can't tell which instrument is asserting SRQ. It must read the status bytes of each instrument, looking for one with bit 7 (SRQ) set. This process is called a serial poll and it is implemented with the POLL command in 4050 BASIC. A typical POLL statement is shown below.

```
POLL Device,Status;7;1,1;2
```

Two numeric variables are specified in the POLL statement (DEVICE and STATUS in our example), followed by a list of GPIB addresses. The addresses are delimited by semicolons. If a secondary address is included it is separated from the primary address by a comma.

The 4052A begins by reading the status byte from the first instrument in the list (address 7), then the second (primary address 1, secondary address 1), then the third (address 2), and so on until the device that is asserting SRQ is found (indicated by bit 7 being set in the status byte). When the device that is asserting SRQ is found, its position in the address list is returned in the first variable (DEVICE), and its status byte is returned in the second variable (STATUS).

Normally, the POLL statement is executed as part of an interrupt service routine specified by an ON SRQ statement. The program in Fig. 4-6 illustrates a simple case of an SRQ handling routine called by an ON SRQ statement.

Lines 10-40 set up a three-element array that contains the device addresses. The first device is set for primary address 7, the second is set for address 1, and the third is set for address 2. Then, line 50 tells the 4052A to GOSUB to line 200 when an SRQ occurs on the GPIB. Line 60 causes the 4052A to suspend program execution until an interrupt occurs.

```
10 DIM Device_nums(3)
20 Device_nums(1)=7
30 Device_nums(2)=1
40 Device_nums(3)=2
50 ON SRQ THEN 200
60 WAIT
70 GOTO 60
     :
     :
200 POLL Device,Status;7;1,1;2
210 PRINT "SRQ FROM DEVICE NUMBER ";
220 PRINT Device_nums(Device);" STATUS: ";Status
230 RETURN
```

Fig. 4-6. *A simple SRQ handling routine called by an ON SRQ statement.*

When the SRQ occurs, the 4052A GOSUB's to line 200 and begins polling the devices. The first device found asserting SRQ causes the polling process to stop. The POLL command returns the status byte of this instrument in STATUS and the index into the address list in DEVICE. The value returned in DEVICE selects the element of array DEVICE_NUMS that contains that instrument's address and lines 210 and 220 print the address and status byte. Finally, line 230 returns control to the loop at line 70, where it jumps to line 60 and waits until another SRQ occurs.

Status byte format. Status bytes returned by Tektronix instruments fall into two categories: system status bytes and device status bytes. System status bytes define conditions that are common among all instruments that conform to the Tektronix Codes and Formats Standard. Device status bytes define conditions that may be unique to the type of instrument.

System status bytes are further divided into normal and abnormal system status. Normal conditions include power up and operation complete. Abnormal conditions include command error and internal error. These two types of status bytes are differentiated by the state of bit 6.

In general, the status byte contains the following information:

BIT 8 - System status=0
Device status=1

7 - SRQ not asserted=0
SRQ asserted=1

6 - Normal condition=0
Abnormal condition=1

5 - Not busy=0
Busy=1

4 - Encoded device/system status

3 - Encoded device/system status

2 - Encoded device/system status

1 - Encoded device/system status

The system status bytes and device status bytes and their meanings are defined in the instrument manuals.

Processing the status byte. Once the status byte for an instrument has been read, the 4052A may need to take some action based on what the status byte says. Status byte processing for Tektronix instruments can be broken into two major parts—processing system status bytes and processing device status bytes.

The meaning of system status bytes is common to all Tektronix instruments; they can be processed without regard for the specific instrument that generated them. For example, a decimal 97 status byte means that an instrument has received a command that it does not understand.

System status bytes all have a zero in bit 8, so their decimal value is 127 or less. This provides a convenient means of testing whether a status byte is a system status or device status. If the byte is a system status byte, it can be processed by a common routine for all instruments. If it is a device status byte, separate routines that handle the device-specific status bytes are called.

```
200 POLL Device,Status;7,1
210 PRINT "SRQ FROM INSTRUMENT ";
220 IF Status>=128 THEN 500
230 IF Status<96 THEN 250
240 Status=Status-29
250 Status=Status-64
260 GOSUB Status OF 280,300,320,340,360,380,400,420,440
270 RETURN
280 PRINT "** POWER UP **"
290 RETURN
300 PRINT "** OPERATION COMPLETE **"
310 RETURN
320 PRINT "** USER REQUEST **"
330 RETURN
340 PRINT "** COMMAND ERROR **"
350 RETURN
360 PRINT "** EXECUTION ERROR **"
370 RETURN
380 PRINT "** INTERNAL ERROR **"
390 RETURN
400 PRINT "** POWER FAIL **"
410 RETURN
420 PRINT "** EXECUTION WARNING **"
430 RETURN
440 PRINT "** INTERNAL WARNING **"
450 RETURN
500 REM ** DEVICE DEPENDENT STATUS PROCESSING STARTS HERE **
```

Fig. 4-7. *A routine for processing system status bytes from Tektronix instruments.*

The program starts in line 200 by polling the instrument at primary address 7, secondary address 1. A single instrument is assumed, but the same technique could be expanded to handle several instruments. Line 210 prints the first half of the message. The semicolon at the end of the PRINT statement inhibits the carriage return usually added to the end of the print statement so that the rest of the message can be printed on the same line.

Then, line 220 checks that the status byte is a system status. If it is greater than or equal to 128 (decimal), it is a device dependent status, and separate routines are called to process the byte. These routines could be added starting at line 500.

Line 230 separates the byte into normal condition and abnormal condition. If the status byte is less than 96, it is a normal condition system status. Line 250 subtracts 64 from the status byte to reduce the byte to a number between 1-3. If the status byte is greater than or equal to 96, line 240 subtracts 29 from it and line 250 reduces it to a number between 4-9. The resulting value is used as an index to select one line number from the list in the GOSUB statement.

Assume, for instance, that the instrument has just been powered up. When line 200 is executed the power up status byte (65) is returned in Y. Line 210 prints the first half of the message. Since the status byte is less than 128, the condition of line 220 is not satisfied, and the GOTO 500 is not executed. Instead, line 230 is executed. The byte is less than 96, so control is passed to line 250, where 64 is subtracted from the status byte. The result is 1 so the GOSUB statement in line 260 sends control to the first line number in the list, line 280. Line 280 prints the last half of the message—** POWER UP **. The complete message printed on the terminal is:

```
SRQ FROM INSTRUMENT ** POWER UP **
```

Processing device-dependent status. Since the device-dependent status bytes are often unique to each instrument, individual routines are usually required to process the status bytes. When a status byte is determined to be device-dependent (greater than or equal to 128), individual processing routines can be called, based on which instrument generated the interrupt.

The simplest method of calling the individual routines is to use a computed GOTO or GOSUB statement. The first variable returned from the POLL statement can be directly used as the index for the GOTO or GOSUB statements. Consider, for example, the program segment shown in Fig. 4-8.

Four instruments are polled in line 10. If the status byte is a system status byte, control is passed to line 400 regardless of which instrument generated the SRQ. If the status byte is a device-dependent byte, the value returned in DEVICE indicates which instrument was asserting SRQ. Line 30 uses this value to select the interrupt handling routine for that instrument. Notice that similar instruments may use the same routine, as the first and third instruments do in this example.

Using the WAIT statement. The WAIT statement in 4050 BASIC provides a way of temporarily

```
10 POLL Device,Status;5;10;7;2
20 IF Status<128 THEN 400
30 GOSUB Device OF 100,200,100,300
40 RETURN
    :
100 REM PROCESS STATUS BYTES FROM INSTRUMENTS #1 AND #3
    :
200 REM PROCESS STATUS BYTES FROM INSTRUMENT #2
    :
300 REM PROCESS STATUS BYTES FROM INSTRUMENT #4
    :
400 REM PROCESS SYSTEM STATUS BYTES
```

Fig. 4-8. *The computed GOTO and GOSUB statements make calling individual device-dependent status processing routines simple. This routine illustrates the use of a computed GOSUB.*

suspending execution of a program while waiting for an interrupt to occur. An ON statement should be executed for the interrupt. When the interrupt occurs, control is transferred to the line number specified in the ON statement.

The WAIT statement can be used to synchronize instrument and controller operations. For example, many acquisition instruments generate an SRQ when an acquisition sequence completes. The WAIT statement can be used to delay reading the data from an instrument until the acquisition-complete interrupt occurs. The program in Fig. 4-9 illustrates this technique used with a 492P Programmable Spectrum Analyzer.

```
10 ON SRQ THEN 100
20 Eos=0
30 PRINT @1:"SIGSWP;EOS ON;SIGSWP"
40 WAIT
50 IF Eos=0 THEN 40
60 PRINT @1:"CURVE?"
70 INPUT _data
   :
   :
100 POLL Device,Status;1
110 IF Status<>66 THEN 130
120 Eos=1
130 RETURN
```

Fig. 4-9. *A sample program using WAIT to synchronize the 4052A with a 492P Programmable Spectrum Analyzer.*

The program sets the 492P to single sweep mode, turns on the end-of-sweep interrupt and arms the sweep in line 30. Line 40 suspends execution of the program until an interrupt occurs. When it does, control is transferred to line 100. The instrument is polled and the status byte is tested. If the end-of-sweep status (66) is returned, a flag variable, EOS, is set to 1. Any other status byte causes the flag to remain zero.

After the interrupt is serviced, control returns to line 50. If the flag is set, indicating that the end-of-sweep interrupt occurred, execution continues. Otherwise, control is returned to the WAIT statement. Lines 60 and 70 send the CURVE? query and read the data into array _DATA.

Using GPIB Peripherals

A variety of GPIB-interfaced peripheral devices are also available for program and data storage, graphic and alphanumeric output, and data logging. Tektronix manufactures several GPIB peripheral devices particularly designed for compatibility with the 4052A. When these devices are used, the mechanics of addressing and controlling the peripherals are handled automatically with high-level 4050 BASIC statements.

The 4052A's powerful I/O system allows you to address Tektronix GPIB peripherals with the same high-level BASIC statements used to address internal devices, such as the graphic display or tape drive. All you have to do is specify the peripheral device number of the external peripheral device in the I/O statement. For example, to draw a line on a GPIB-interfaced 4662 plotter, use the statement:

```
DRAW @1:X,Y
```

This statement draws a line on the plotter from the current pen position to the coordinates specified in X and Y. The only difference between this statement and the equivalent statement for for 4052A's internal graphic display is the peripheral device number.

This section takes a brief look at four Tektronix GPIB peripherals and their operation with the 4052A. More complete information on their operation is contained in their Operators manuals.

4907 Flexible Disk File Manager. One such peripheral device is the Tektronix 4907 File Manager. The 4907 provides fast random access flexible disk storage for 4050-series Desktop Computers. Up to 630K bytes of program and data storage is available on each disk with up to three disks per system. The 4907 comes with a special ROM pack that adds several new commands to the 4052A's vocabulary. These commands provide the following functions:

- File naming
- File security with passwords
- Automatic increase in file space when necessary
- File copying
- Multiple file access
- Recording time and date of all file activities
- File renaming
- Five file storage levels
- Fast random access files

Like the other 4050 ROM packs, the 4907 operating system software occupies no RAM memory, so space is left free for user programs and data. The easy-to-learn, plain English commands also make programming the 4907 a simple task. For example, to create a new file, use the CREATE command. A typical CREATE command is shown below.

```
CREATE "FILE.JNK";100,128
```

This command creates a file named FILE.JNK in the current library with 100 records of 128 bytes/record. (For more information on the CREATE command, refer to the 4907 Operator's Manual.)

The 4907 also provides five levels of file storage through the use of "libraries." A library contains the names of other files or libraries, and it is used to group files. Figure 4-10 shows the simplest form of file structure on a 4907 disk. This disk contains a single level of files. No libraries are used.

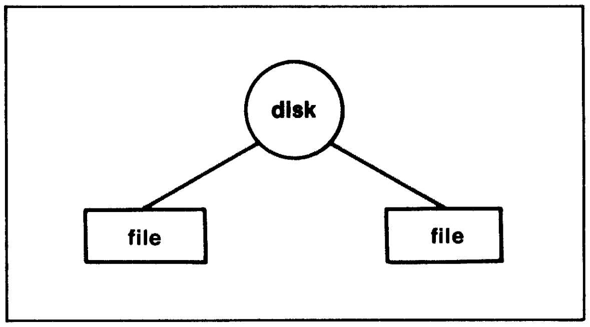


Fig. 4-10. *The simplest file storage structure on the 4907 contains only a single level with no libraries.*

When a library is added, files are placed on the next lower level, as shown in Fig. 4-11. This structure can be extended for up to five levels as shown in Fig. 4-12.

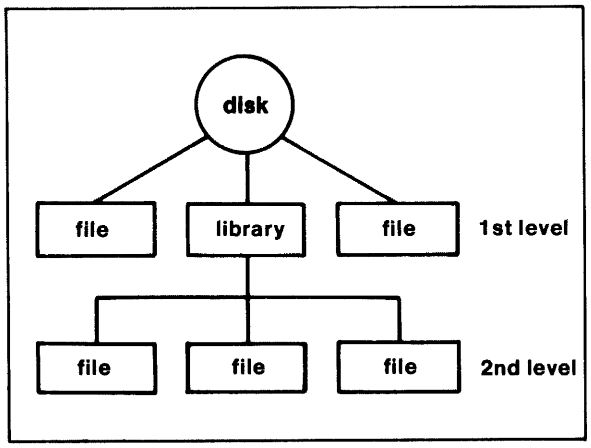


Fig. 4-11. *Libraries are used to group files on the subsequent levels. Files and libraries may be mixed on any level except the last.*

This five-level storage structure allows an almost limitless variety of file storage arrangements to meet your individual needs and make data access easier and more logical.

4924 Digital Cartridge Tape Recorder. The 4924 is a general purpose cartridge tape drive with a GPIB interface. It uses the same tape cartridge and records data in the same format as the 4052A's internal tape drive. Thus, the 4924 can be used as an extension of the internal tape.

The 4924 can be operated in two basic modes. The first mode employs commands issued over the GPIB. Secondary addresses are used to send commands as previously discussed under **I/O Addressing in 4050 BASIC**. Since the command codes are the same for the 4924 and the internal 4052A tape drive, the interface is simple. For example, to kill a file on an external 4924, use the KILL command:

```
KILL @2:5
```

This command kills file number 5 on a 4924 set for primary address 2. The 4052A automatically sends the secondary address that tells the 4924 to execute a KILL operation.

In addition, commands can be sent to the 4924 using device-dependent ASCII messages in place of the secondary address. For example, to execute the KILL command described above, the controller sends the command primary address (separate command and data primary addresses are used in this mode). Then, with attention unasserted, the letter "K" is sent followed by a delimiter (space, comma, or semicolon), the file number, and a carriage return. Finally, the UNListen message is sent with attention asserted.

In the second mode, the 4924 operates manually from the front panel. The front-panel buttons are used to perform basic tape operations such as advance forward or reverse, talk, or listen. This mode is most useful when the 4924 is operated as a data-logging device without a controller.

4662 Interactive Digital Plotter. The Tektronix 4662 is an interactive plotter with RS-232C and GPIB interfaces. The plotter can print alphanumerics and graphics on paper or other media up to 11 by 17 inches. In addition, it can perform as a graphic input device.

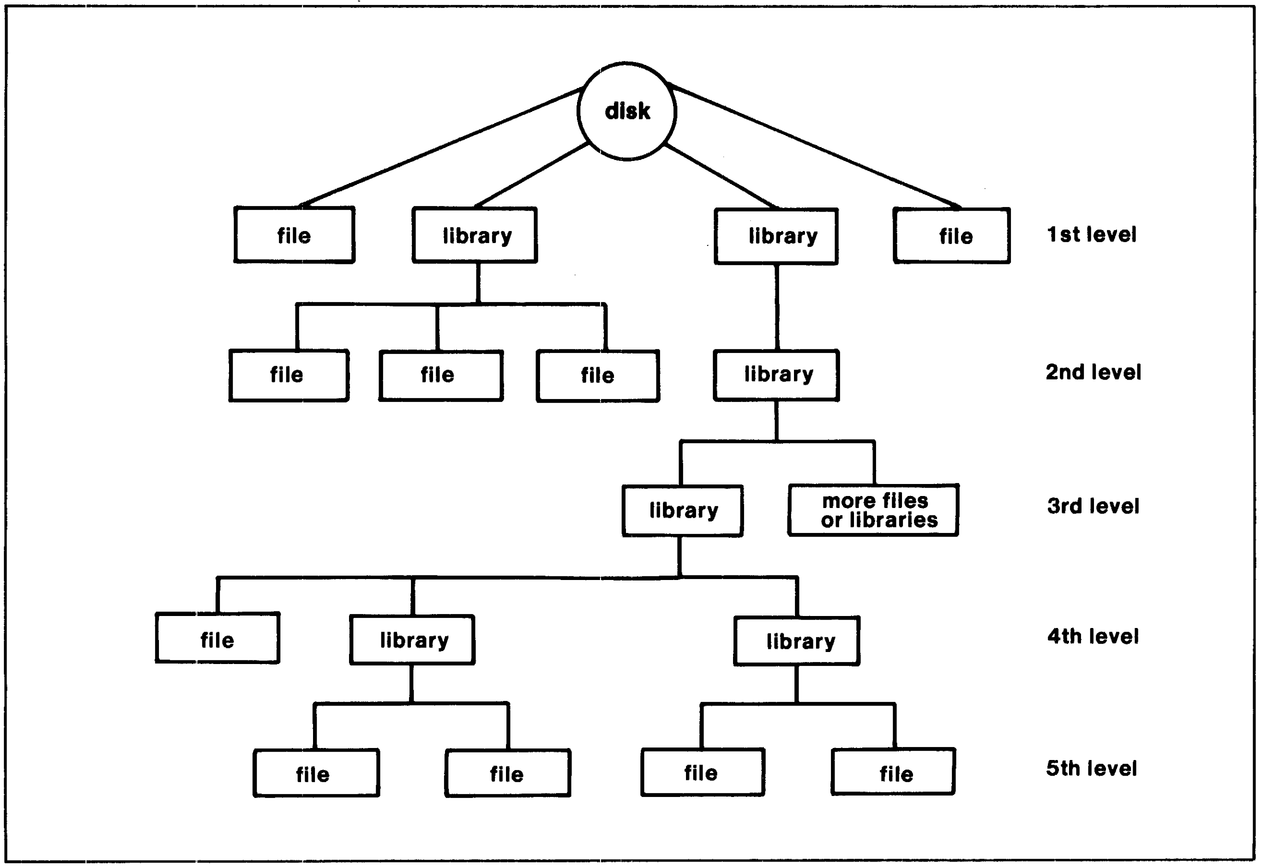


Fig. 4-12. *The 4907 file structure can be expanded to five levels. Files and libraries can be mixed on any level except the fifth.*

When interfaced via the GPIB, the 4662, like the 4924 tape drive, accepts commands in one of two modes. In the first mode, commands are sent as secondary addresses. The command codes correspond to those used in the 4052A, so the plotter can be fully controlled using simple high-level BASIC statements. For example, to move the plotter pen to a specific location, execute the MOVE command:

```
MOVE @1:X,Y
```

where: X is the horizontal coordinate in Graphic Display Units; Y is the vertical coordinate in Graphic Display Units.

4050 BASIC includes statements to execute relative and absolute MOVEs and DRAWs, to generate axes, rotate and scale alphanumerics, and to window and scale graphic data.

The 4662 also implements another command mode for interfacing with other controllers. In this mode, device-dependent messages are used to send plotter commands. Normally, this mode is not used with 4050-series controllers, since the secondary address scheme is implemented automatically in 4050 BASIC. The 4662 Operator's manual contains complete information on using this mode.

4956 Graphic Tablet. The 4956 Graphic Tablet converts pen position, on its platen, to horizontal and vertical coordinates. You can use it to digitize drawings or schematics by placing the article to be digitized on the platen and tracing it with the pen provided.

To obtain the horizontal and vertical coordinates, you simply use the INPUT statement:

```
INPUT @1:X,Y,Status$
```

After this statement executes, STATUS$ will contain a status byte that gives you additional information about the state of the graphic tablet.

Section 5 — Processing and Displaying Data

Most GPIB systems make some type of measurement as part of their job, whether it be a simple voltage measurement or a complete waveform acquisition. The first step in making the measurement is to acquire the signal and convert it to a digital format. That is the job of the acquisition instrument (i.e., waveform digitizer, digital voltmeter, etc). The output of this instrument is then sent to the controller over the GPIB (Fig. 5-1).

Once data is acquired and transferred to the 4052A, some processing is often required to derive the desired information. It's important to remember that the 4052A is processing a digital representation of the input signal—a string of numbers—not the signal itself. The numbers usually represent vertical signal amplitudes at discrete sample points along the signal. The position of each number in the series represents its horizontal time location on the signal (Fig. 5-1).

When the data is processed, the 4052A manipulates only the numbers stored in its memory. Almost without exception, the processing is done on an element-by-element basis, starting with the first element in the array and progressing to the last one.

For example, let's say you have acquired a waveform, and it has been transferred into an array in the 4052A. Now, maybe you want to add a constant to it. The data might represent a voltage waveform to which you want to add a four-volt bias. The 4050 BASIC statement is:

```
Waveform=Waveform+4
```

When this statement is executed, the 4052A adds four to the first element in array WAVEFORM. Then it adds four to the second element, and the third, and so on until all the elements have been processed.

This same element-by-element process is also used in subtracting a constant from an array, multiplying an array by a constant, or most any other arithmetic operation. It is also used when two arrays are processed in a statement, such as multiplying two arrays. For example, the statement

```
A=B*C
```

causes the 4052A to multiply each element of array B by the corresponding element in array C, and store the result in array A.

This all seems so simple. And it is—if you avoid the more common pitfalls by keeping the following DOs and DON'Ts in mind:

DON'T attempt to combine (add, subtract, multiply, or divide) arrays of different lengths since the element-by-element processing won't complete. If you attempt such an operation, the 4052A will remind you by printing an error message.

DON'T attempt to combine data arrays acquired at different sampling intervals. For example, don't add a waveform array that was acquired at a 5-microsecond sampling interval with a waveform sampled at 15 microseconds/sample. The time scaling may lead to erroneous or confusing results.

DO be cautious of dividing by zero or very small numbers (such as occur at zero crossings on repetitive waveforms) since this can lead to SIZE errors or erroneous results.

DO keep in mind the limits of the 4052A's calculation accuracy. All math operations are computed to 14 digits of accuracy, so for all but the most lengthy calculations, the round-off error is insignificant. But, it's important to remember that round-off error can accumulate in lengthy calculations until it becomes significant.

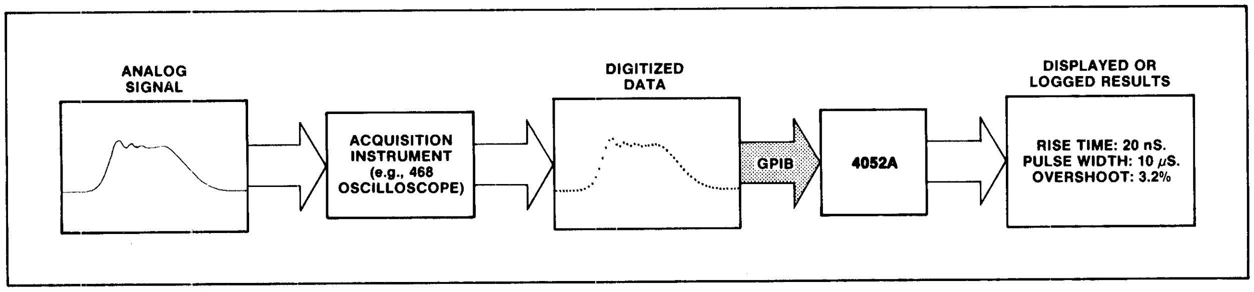


Fig. 5-1. *Automated measurement begins with the acquisition instrument. It sends digitized data to the 4052A. The data may either be logged or processed and the results logged or displayed.*

Using the ROM pack routines. Most signal processing needs go beyond simple mathematical combinations of constants and waveforms. Often, special signal processing or array operations are required, such as finding the maximum or minimum of an array, integrating, differentiating, or other complex operations. To facilitate this type of processing, many of the common signal and array processing functions are provided in a series of ROM (Read-Only Memory) packs that plug into the slots in the rear of the 4052A.

The ROM pack commands execute much faster than equivalent programs written in BASIC. Some commands execute as much as ten times faster! Also, the ROM pack commands do not occupy any user memory space.

ROM pack routines are executed with a simple CALL statement. The syntax of the statement is

```
CALL "routine name"[,parameter...]
```

For example, to find the maximum value in an array, use the 4052R07 ROM call, MAX. The MAX routine can be called with this statement:

```
CALL "MAX",A,_value,_index
```

Where A is the source array that is searched. The maximum value of the array is returned in _VALUE, and the location (subscript) in the array of the maximum value is returned in _INDEX.

The routine name may also be stored in a string variable and substituted for the name in the CALL statement, as illustrated below.

```
MAX$="MAX"
   :
   :
CALL MAX$,A,_value,_index
```

Signal Processing ROM Packs. The Signal Processing ROM Packs (4052R07 and 4052R08) provide 15 common waveform and array processing functions. The functions are listed and briefly described below.

4052R07 Signal Processing ROM Pack No. 1

MAX—finds the maximum of an array.

MIN—finds the minimum of an array.

CROSS—finds the location of a specified crossing level within an array.

DIF2—performs a two-point differentiation of an array.

DIF3—performs a three-point differentiation of an array.

INT—integrates an array

DISP—displays a graph of an array in raw form (without axes).

4052R08 Signal Processing ROM Pack No. 2

FFT—computes the fast Fourier Transform of a one-dimensional array.

IFT—computes the inverse Fourier transform of a one-dimensional array.

CONV—convolves two one-dimensional arrays.

CORR—correlates two one-dimensional arrays.

POLAR—converts an array of complex data from rectangular form (real and imaginaries) to polar form (magnitude and phase).

TAPER—multiplies an array by a cosine window of program-selectable weights.

UNLEAV—sorts an array of interleaved FFT data into two arrays, one containing real and one containing imaginary components.

INLEAV—interleaves the real and imaginary data from two input arrays into a third array whose format is acceptable to the IFT command.

Real-Time Clock ROM Pack. In many real-time applications, such as automated testing, the time when an event occurs can be as important as the event or measurement itself. The 4052R09 Real Time Clock ROM Pack provides time and date, elapsed time, and vectored time interrupt capabilities for the 4052A. Commands provided by this ROM pack are:

SETIME—sets the clock to the desired time and date.

RDTIME—reads the time and date.

STARTW—resets and starts the stopwatch incrementing in 0.1 second steps.

STOPIT—reads the elapsed time from the stopwatch.

ONTIME—sets the programmable interrupt delay. When the delay expires, control is transferred to line 84 of the user program.

Graphing data. The old adage "a picture is worth a thousand words" may be overworked but it's still true—particularly in the realm of science and engineering. An important relationship involving two or more variables may be difficult, if not impossible, to understand when presented as a column of numbers. Yet, by means of a graph or chart, the same relationship can often be recognized at a glance.

In addition, ideas or information that are difficult to convey in words can often be easily conveyed in pictures. For example, describing a test set-up in words can be difficult, especially when low-skill operators are involved. But a picture of the test set-up provides the same, if not more, information for the operator in simple terms that are easily understood.

The 4052A's powerful, high-resolution graphics capability and extended BASIC language make generating and displaying graphics a simple task. Statements are included in 4050 BASIC to scale data, map it into a defined window, move the window anywhere on the screen, and perform a variety of graphic functions within that window.

Data can be defined in any units appropriate to the application. For example, a program that tests frequency response of a system might use decibels for vertical axis units and frequency for the horizontal axis units. Once these units are declared and their limits defined, the 4052A automatically maps the user's data units into the graphic display.

Sample program. The program in Fig. 5-2 illustrates the use of several signal processing ROM pack functions as well as some simple graphics functions. It performs some basic pulse analysis on a waveform stored in array A. The program assumes a simple pulse is stored in array A and that its horizontal scale factor is stored in H.

The output of the program is a graph (without axes) of the input waveform and a pulse parameter summary printed below the graph.

The program begins by setting the base of the pulse equal to zero to simplify processing. Line 110 finds the minimum value and line 120 subtracts this value from the entire waveform, to set the base of the pulse to zero. Line 130 returns the maximum value of the pulse in _MAX. Then, lines 140 and 150 use this maximum value to find the 90% and 10% amplitude points and return the location of these points in T90 and T10, respectively. The difference between these points, multiplied by the horizontal scale factor (H), is the rise time for the pulse. Lines 170-190 do the same thing for the fall time.

Line 200 finds the first 50% point and line 210 finds the second 50% point. The difference between these

```
100 REM 4052A BASIC PULSE ANALYSIS
110 CALL "min",A,_min,Index_min
120 A=A-_min
130 CALL "max",A,_max,Index_max
140 CALL "cross",A,0.9*_max,T90
150 CALL "cross",A,0.1*_max,T10
160 Risetime=(T90-T10)*H
170 CALL "cross",A,0.9*_max,T90,2
180 CALL "cross",A,0.1*_max,T10,2
190 Falltime=(T10-T90)*H
200 CALL "cross",A,0.5*_max,T50_1
210 CALL "cross",A,0.5*_max,T50_2,2
220 Width=(T50_2-T50_1)*H
230 REM GRAPH RESULTS
240 VIEWPORT 10,70,40,100
250 WINDOW 1,UBOUND(A,-2),0,_max
260 CALL "disp",A
270 VIEWPORT 0,130,0,100
280 WINDOW 1,130,0,100
290 MOVE 1,25
300 PRINT "RISE TIME=";Risetime
310 PRINT "FALL TIME=";Falltime
320 PRINT "50% WIDTH=";Width
330 END
```

Fig. 5-2. *A sample program to compute some basic pulse parameters on a waveform stored in array A. The routine also graphs the waveform without axes.*

points multiplied by the horizontal scale factor is the 50% pulse width.

Line 240 reduces the size of the viewport to leave room for the pulse parameter information to be printed below. The WINDOW statement sets the limits of the data to be graphed. Line 260 displays the waveform in the current WINDOW and VIEWPORT.

Finally, lines 270-320 reset the VIEWPORT and WINDOW and print the pulse parameters below the graph. Figure 5-3 shows a sample output from the program.

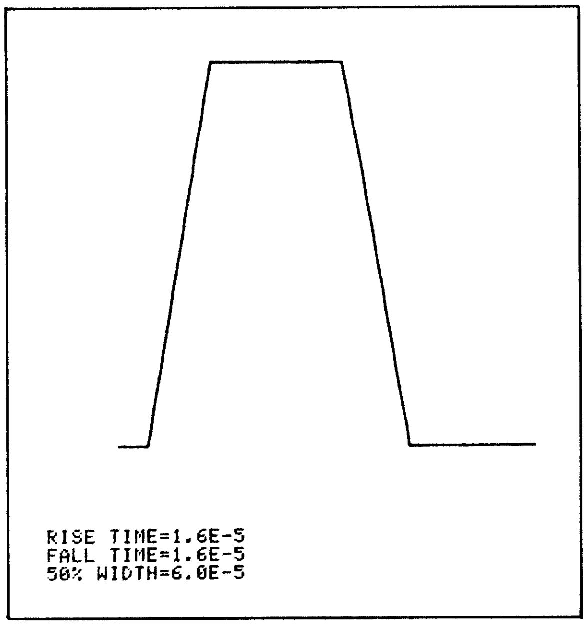


Fig. 5-3. *Sample output from the pulse analysis program.*

Section 6 — Estimating GPIB System Performance

One of the most frequently asked questions about any GPIB system is "how fast will it go?" This question is often critical to the design and implementation of a system. Yet, it is often very difficult to answer. The complete performance picture is composed of a multitude of parts, many that are difficult to estimate. However, a good understanding of the basic factors that contribute to the performance of the system will help develop a good estimate of the system's overall performance.

GPIB system performance is affected by several factors. The key factors are:

Data transfer time
Processing time
Data acquisition time
Human interaction time

Data Transfer Time

A certain amount of time is required for messages to be transferred across the GPIB. This time is called the data transfer time. It includes time to transfer interface messages as well as device dependent data across the bus.

The asynchronous bus. The GPIB is an asynchronous bus. That is, data is transferred at a rate determined entirely by the instruments on the bus; there is no clock signal. The maximum data transfer rate is determined by the slowest device involved. When the talker or controller places a byte on the bus, all listeners must accept the data byte (indicated by releasing the $\overline{\text{NDAC}}$ line) before the talker can proceed to the next byte. Any listener can delay the transfer simply by holding $\overline{\text{NDAC}}$ low (asserted).

This asynchronous bus allows a variety of instruments with different speeds to work together. But, it also means that any slow device involved in a transfer slows the entire transfer down to its rate. To illustrate, consider the system shown in Fig. 6-1. The maximum data transfer rate for each device is shown in the figure.

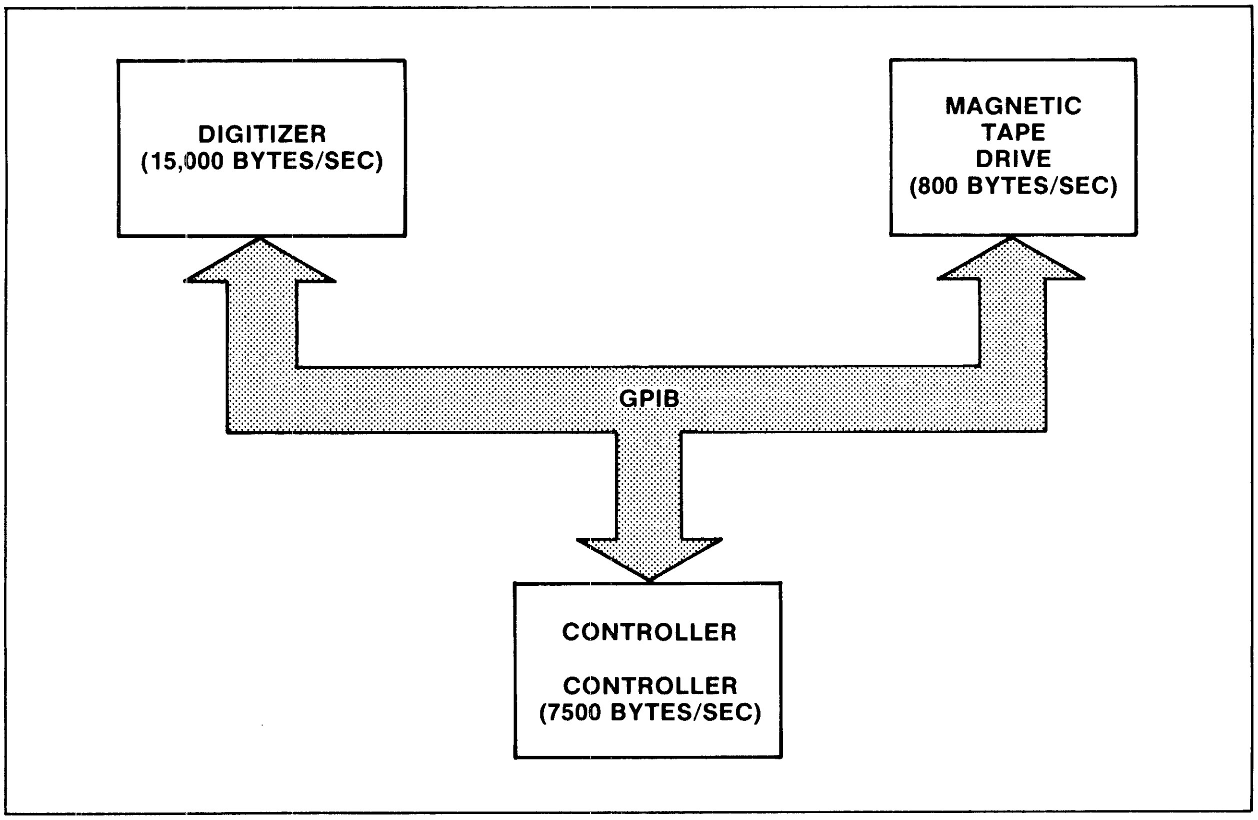


Fig. 6-1. *A typical GPIB system showing the maximum data transfer rates for each device in the system. All transfers are limited to the speed of the slowest device involved.*

When data is transferred between the digitizer and tape drive, the transfer is limited to 800 bytes/second by the tape drive. When data is transferred between the controller and digitizer, the transfer can proceed at up to 7500 bytes/second, since the tape drive is not involved. In no case will the full speed capability of the digitizer be realized, since the other two devices limit the speed of the transfers.

In practice, the data transfer rate is usually **lower** than the rate of the slowest listener. This is because the bulk of the processing performed by the output device must be done **before** data is transferred, whereas the input device must do most of its processing **after** it receives data. The two processes, therefore, can not be done in parallel but must be done sequentially.

GPIB data transfer timing. When a 4052A I/O statement is used to transfer data on the GPIB, up to five events occur, though they don't all occur in every I/O statement. These events are graphically represented in Fig. 6-2.

STATEMENT OVERHEAD—Every I/O statement has statement overhead. When an I/O statement is executed, the 4052A first examines the statement for content and syntax. For high-level I/O statements, such as PRINT or INPUT, the I/O address is checked to see if default values are necessary (e.g., when no secondary address is specified). The parameter list is also examined to see if string variables, string or numeric constants, numeric variables, or array variables are specified. If numeric expressions are specified, they must be reduced to constants before going on. The statement overhead time is variable and depends entirely on the type of statement and its parameter list.

ADDRESSING SEQUENCE—Every I/O statement except the low-level RBYTE and WBYTE has an addressing sequence. First, the 4052A asserts ATN and issues the UNTalk interface message followed by the absolute primary talk or listen address for the peripheral device specified. The secondary address is sent next, unless secondary address 32 is specified (secondary address 32 tells the 4052A to send no secondary address). After the primary and secondary addresses are sent, ATN is released.

DATA TRANSFER—For ASCII data transfers, the actual data transfer occurs in segments called data bursts. Data is transferred at the maximum rate determined by the slowest device involved in the transfer. On input operations, the 4052A receives bytes from the talker and stores them in an intermediate buffer memory (called the I/O buffer). On output operations, data is transmitted from the I/O buffer over the bus to the listener(s).

Binary data transfers are done two different ways. With RBYTE and WBYTE, each byte is processed as it is transferred. With READ and WRITE, no processing is necessary; data is moved directly to or from 4052A memory. Both binary transfer methods result in continuous data transfers (no data bursts).

BUFFER OVERHEAD—The I/O buffer can hold a maximum of 72 characters. If an ASCII data transfer involves more than 72 characters, the transfer is performed in 72-character bursts. On output, the 4052A converts the first 72 characters from internal binary format to the correct format for transmission and loads the formatted data into the I/O buffer. Then, the contents of the I/O buffer are transmitted.

On input, 72 characters are received and loaded into the I/O buffer. Then, this data is converted to the correct format for the variables specified in the statement and the data is stored.

Between each burst of input or output, a variable amount of time is required for refilling the I/O buffer on output or emptying it on input. This operation is called buffer overhead and the amount of time required depends entirely upon the type and amount of data.

UNADDRESSING SEQUENCE—Every I/O statement except RBYTE and WBYTE has an unaddressing sequence at the end of the data

Statement Overhead	**Addressing Sequence**	**Data Burst**	**Buffer Overhead**	**Unaddressing Sequence**

Fig. 6-2. *GPIB data transfers are composed of five parts, each contributing to the total data transfer time.*

transfer. The 4052A asserts ATN, sends the universal commands UNTalk and UNListen, and releases ATN.

Table 6-1 lists the GPIB timing information for the PRINT, INPUT, WBYTE, RBYTE, WRITE, and READ statements. The rates given are the **maximum** rates for the 4052A. The information presented in the table is discussed in more detail in subsequent subsections.

Estimating data transfer rate for PRINT. Because of the versatility and many variations in PRINT statements, the statement overhead times are very difficult to calculate. The time can vary from about 3 milliseconds to a few tens of milliseconds. For simple variables (not expressions), add about 2 milliseconds per variable to the 1.5-millisecond minimum to get a rough estimate of the statement overhead time.

The addressing period for PRINT is constant, regardless of the format of the statement. Unless the secondary address is suppressed, the addressing period takes about 1.7 milliseconds (minimum).

TABLE 6-1
4052A GPIB TIMING INFORMATION

	Statement Overhead	**Addressing Sequence**	**Data Transfer Rate**	**Buffer Overhead**	**Unaddressing Sequence**
PRINT	1.5 ms + 1-4 ms/element	1.7 ms with secondary 1.36 ms primary only	111,000 bytes/sec burst rate 9 μs/byte	strings 285 μs + 86 μs/char numerics integers ≈3.0 ms/sample non-integers ≈3.4 ms/sample	1.35 ms
INPUT	1.9 ms + 180 μs/variable	1.82 ms with secondary 1.42 ms primary only	71,000 bytes/sec burst rate 14 μs/byte	strings 395 μs + 37 μs/char numerics 600 μs/sample + 58 μs/digit	1.35 ms
WBYTE	0.75 ms + 150 μs/element	N/A	array 1646 bytes/sec 608 μs/byte single element 1500 bytes/sec 665 μs/byte	N/A	N/A
RBYTE	600 μs + 270 μs/element	N/A	array 1481 bytes/sec 675 μs/byte single element 1344 bytes/sec 744 μs/byte	N/A	N/A
WRITE	1.9 ms + 180 μs/element	1.72 ms with secondary 1.34 ms primary only	string 112,000 bytes/sec 8.9 μs/byte numeric 1850 numbers/sec 540 μs/number	N/A	1.35 ms
READ	1.9 ms + 180 μs/element	1.79 ms with secondary 1.50 ms primary only	string 82,700 bytes/sec 12.1 μs/byte numeric 1850 numbers/sec 540 μs/number	N/A	1.35 ms

With the secondary address suppressed, the addressing period takes about 1.4 milliseconds (minimum).

Buffer overhead for strings takes about 285 microseconds plus 86 microseconds for each character. Buffer overhead for numerics is about 3 milliseconds per sample for integers and about 3.4 milliseconds per sample for non-integer numbers. (A sample is one or more digits that make up a value.)

When a buffer of characters is prepared for transmission, the 4052A can transmit them at about 111,000 bytes/second. This, of course, assumes that the listener is significantly faster, and does not affect the transfer rate. With a slower listener on the bus, the overhead time will not be affected, but the maximum data burst rate will be the listener's slower rate.

The unaddressing period takes about 1.35 milliseconds, regardless of the statement format.

Consider this simple print statement:

```
PRINT @2:1;2;3;4;5;6;7;8;9
```

In this example, less than 72 characters are sent, so burst data rate of 111,000 bytes/second applies. Since the default PRINT statement format inserts a space before each number, the samples each contain two bytes. Therefore, 111,000/2 or 55,000 samples can be transmitted in a second. If each sample was a two-digit number, the rate would be reduced to 111,000/3 or 37,000 samples per second.

This same idea applies to numeric arrays transmitted in the form:

```
DIM A(10)
A=1
PRINT @2:A;
```

Here, two-byte samples are transmitted as above at a rate of 55,000 samples/second.

If a comma is substituted between variables or constants in the PRINT statement or the trailing semicolon is left off an array variable, the 4052A formats each element into an 18-character field by adding spaces. Thus, every element, no matter how many digits it contains, takes 18 bytes to transmit. As a result, data samples are transmitted at 111,000/18 or 6167 samples/second.

Notice that simply removing the trailing semicolon from the statement above reduces the data rate from 55,000 samples/second to 6167 samples/second! It's easy to see how a minor change in a statement can drastically affect the data rate.

Estimating data transfer rate for INPUT. The statement overhead for INPUT is a function of the statement format and the number of variables in the statement. The basic statement overhead time is about 1.9 milliseconds plus 180 microseconds for each variable in the statement.

The addressing period for INPUT is essentially the same as the PRINT statement addressing period—1.8 milliseconds with the secondary address; 1.4 milliseconds without the secondary address.

Data bursts up to 72 characters can be received at about 71,000 bytes per second (14 μs per byte). Again, this assumes that the talker can send data at least this fast.

The buffer overhead period depends on the number of characters in the buffer and the type of variable they are destined for. For numeric variables, each variable requires about 600 microseconds for a single digit plus about 58 microseconds per additional digit. So, to convert a single five-digit variable requires about 890 microseconds (600 μs + 5 * 58 μs).

For string variables, the first character takes about 395 microseconds. Each additional character requires about 37 microseconds.

At the end of a transmission, the buffer is emptied whether it is full or not, and the data is assigned to variables.

Finally, it takes the 4052A about 1.35 milliseconds to assert ATN, send the UNTalk and UNListen commands, and release ATN.

For example:

```
10 DIM X(1000)
20 INPUT @4,32:X
```

Device 4 sends 1000 data samples with five digits in each sample. A comma delimits each sample. A typical data stream might look like this:

```
36524,37428,39266,39694...<996 more samples>
```

The time to execute the INPUT statement in line 20 is estimated as shown below.

Statement overhead 1.9 ms + 180 μs ≈	2.08 ms
Addressing period≈	1.35 ms
Actual data transfer time (6000 bytes/71,000)≈	84.5 ms
Buffer overhead:	
600 μs ⋆ 1000 samples + 58 μs ⋆ 5000 digits ≈	890.0 ms
Unaddressing period ≈	1.35 ms
Total data transfer time ≈	979.33 ms
Effective data sample transfer rate ≈	1020 samples/sec.

Estimating data transfer rate for WRITE. The WRITE statement transfers data over the GPIB in 4052A internal binary format. Each data item is preceded by a two-byte header which identifies the data item type (number or string) and the length of the item (in bytes). The length of a numeric data item is always eight bytes plus the header. The length of a character string is one byte per character plus the header.

Since data is sent in 4052A internal binary format, no conversion (and therefore no buffer overhead) is necessary and a WRITE data transfer is very fast. However, because the data is in 4052A internal binary format, WRITE can generally only be used for sending data to a storage device (such as the 4907 File Manager). Data written to a storage device can later be loaded directly back into the 4052A with the READ statement.

The addressing period for WRITE is about the same as any other high-level I/O statement. It takes about 1.7 milliseconds to send a primary and secondary address. The primary address alone takes about 1.3 milliseconds to send.

The data rate for strings is about 112,000 bytes/second (8.9 μs/byte). For numbers, the data rate is about 1850 samples/second.

The unaddressing period for WRITE, like all other high-level I/O statements, takes about 1.35 milliseconds.

Estimating data transfer rate for READ. The READ statement is generally used to bring data in 4052A internal binary format back from a peripheral storage device. It is usually not practical to receive data from an instrument with READ, since most instruments cannot transmit data in the required format.

When receiving numeric data, each sample consists of two bytes of header plus eight bytes of floating-point binary data. The maximum sustained data rate for numeric data is 1850 samples/second (540 μs/sample).

Character strings can be received at 82,700 bytes/second (12.1 μs/byte).

Estimating data transfer rate for WBYTE. The WBYTE statement is normally used in situations where PRINT or WRITE can't be used or where a binary transfer would be faster. It can be used to transmit binary data to an instrument (not 4052A internal binary format), or to set up a transfer between two or more devices. It can also be used to transmit multiline interface messages.

Remember that WBYTE does not perform the automatic addressing and unaddressing functions that PRINT and WRITE do. Therefore, the addressing and unaddressing periods do not apply to WBYTE.

The conversion of each value specified in a WBYTE statement to binary occurs just before the value is transmitted. As a result, the data rate depends on the type of variable. WBYTE can transmit an array at 1646 bytes/sec (608 μs/byte). Single element numerics (not arrays) can be transmitted at 1500 bytes/second.

Estimating data transfer rate for RBYTE. The RBYTE statement is used to receive binary data bytes from the GPIB. Since INPUT receives ASCII data and READ requires 4052A internal binary format, RBYTE is the only choice for binary data sent by some instruments (unless you have a 4052R14 Option 1A GPIB Enhancement ROM Pack).

The statement overhead period for RBYTE can be estimated as follows:

Statement overhead ≈ 0.6 ms + no. of variables ⋆ 0.270 ms

EXAMPLE: RBYTE A,B,C

Statement overhead ≈ 0.6 ms + 3 ⋆ 0.270 ms ≈ 1.41 ms

RBYTE can receive data at about 1344 bytes/second (744 μs/byte). When an array variable is specified, the bytes can be received at 1481 bytes/second (675 bytes/second). These data rates can be sustained for any number of bytes.

Since no unaddressing occurs at the end of the RBYTE statement, the execution is complete as soon as the last byte is received.

Multiline interface message traffic. GPIB message traffic also includes multiline interface messages. These messages include the primary and secondary addresses used to set up device-dependent transfers, as well as other messages that implement a serial poll, device clear, or other functions. Multiline interface messages are always sent with ATN asserted, and all devices on the bus must handshake the message, whether they are addressed or not. Thus, the slowest device on the bus regulates the transfer rate.

Multiline interface messages consist of addresses, universal commands, and addressed commands. The addresses are automatically implemented in high-level I/O statements like PRINT and INPUT. They may also be sent with WBYTE. The most commonly used interface commands are implemented in high-level statements such as POLL and INIT. (POLL implements Serial Poll Enable and Serial Poll Disable; INIT implements Device Clear). The remainder of the commands must be sent with WBYTE. Estimating the transfer rate of these bytes was discussed earlier in this section, under **Estimating the transfer rate for WBYTE**.

The serial poll process (implemented with a POLL statement) is often an important part of interface message traffic. Sometimes it is important to service an SRQ interrupt quickly. You need a way to estimate the time it will take to perform a poll. Figure 6-3 illustrates the process of executing a POLL statement. The times shown in Fig. 6-3 will help you estimate the time required for the 4052A to execute the polling process. The times shown are minimum times. As always, the slowest device on the bus regulates the speed of the transfer.

The following example demonstrates how to estimate the time required for the polling process. Assume you want to know how long it will take to execute the following POLL statement:

POLL X,Y;2;1;7

Assume, further, that the second device in the POLL list is a 7854 Programmable Oscilloscope and that the 7854 is requesting service. The address of the 7854 is 1.

Referring to the timing values given in Fig. 6-3, the time required to execute the POLL statement can be estimated as follows.

Statement overhead and bus initialization =	1.5	ms
Assert ATN and issue UNL and SERIAL POLL ENABLE =	1.4	ms
Address 1st instrument to talk (without sec. addrs.) =	0.5	ms
Get status byte from 1st instrument =	0.6	ms
Test bit 7 of status byte and loop =	0.6	ms
Address 2nd instrument to talk (without sec. addrs.) =	0.5	ms
Get status byte from 2nd instrument =	0.6	ms
Test bit 7 of status byte, send UNT and SERIAL POLL DISABLE =	1.35	ms
Total time to execute serial poll =	7.05	ms

Processing Time

Another significant factor in GPIB system performance is processing time—the time required by the controller and instruments to execute commands and process data. It's important to remember that in a GPIB system at least two programs are running. One program is running in the 4052A, and another is running in each of the programmable instruments in the system. Each of these programs take time to execute, and that time must be accounted for in making an estimate of the system's performance.

Controller processing. In most systems, the controllers job goes far beyond simply setting up bus transfers, and collecting data. A significant amount of processing may be required in some applications to extract the desired results from the raw acquired data.

The controller processing time can be broken into two major parts:

I/O Processing
Data processing

The first of these tasks, I/O processing, involves the transfer of data over the GPIB. Estimating the data transfer time is discussed in detail earlier in this section.

The data processing task is as varied as are the applications for GPIB systems. The time required to perform the processing depends on the length and complexity of the program, the speed of the controller, and the number of tasks it is assigned (e.g., how many instruments are on the bus that may interrupt the program, etc). Since these factors are very difficult to quantify, the most practical method of measuring this performance is usually actual testing. A number of programming hints are given in the next section to help improve the performance of 4052A GPIB systems.

Instrument processing. Most programmable GPIB instruments have a microprocessor system inside that controls the GPIB and internal instrument

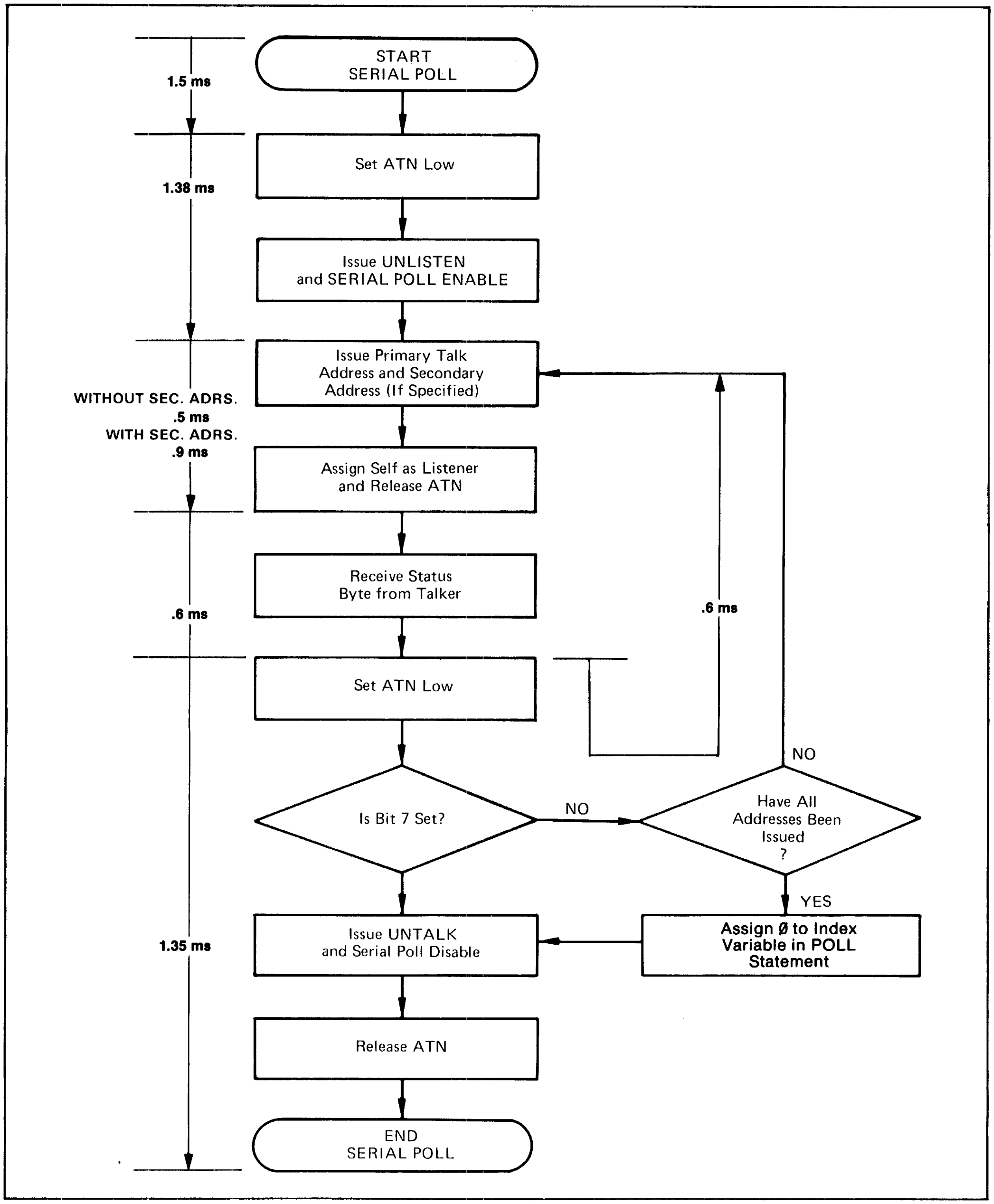

1.5 ms
START SERIAL POLL
Set ATN Low
1.38 ms
Issue UNLISTEN and SERIAL POLL ENABLE
Issue Primary Talk Address and Secondary Address (If Specified)
WITHOUT SEC. ADRS. .5 ms
WITH SEC. ADRS. .9 ms
Assign Self as Listener and Release ATN
Receive Status Byte from Talker
.6 ms
.6 ms
Set ATN Low
NO
Is Bit 7 Set?
NO
Have All Addresses Been Issued ?
YES
Issue UNTALK and Serial Poll Disable
1.35 ms
Assign Ø to Index Variable in POLL Statement
Release ATN
END SERIAL POLL

Fig. 6-3. *A flow chart of the POLL process. Approximate timing values are shown as an aid to estimating the execution time.*

functions. This microprocessor has three major tasks, though not all of these tasks are implemented in every instrument.

GPIB control and message processing
Instrument control
Data handling

When an instrument receives a message from the GPIB, the internal microprocessor goes to work decoding the message, checking it for errors, and taking the appropriate action. This processing takes time. However, the amount of time is often not specified in instrument manuals. As a result, the only practical way to estimate the time is by direct measurement.

The second task, instrument control, involves accepting and processing front-panel input and monitoring and controlling internal instrument functions.

When the instrument is set-up and the measurement taken, there may be some local processing required to get the data ready for transmission to the controller. For example, code conversions may have to be performed. In some instruments, local processing such as signal averaging or interpolation may be performed. Again, this processing takes time. If the instrument is capable of sending raw data and processed data, comparing the time required to do each task provides a reasonable estimate of the internal processing time.

Data Acquisition

The third major component in system performance is the time required to actually acquire data. In some instruments this may be an insignificant amount of time. In others, such as waveform digitizers, data acquisition can be the most time consuming operation in the system. At least two factors must be considered in data acquisition—triggering and digitizing time. And, in some applications a third component must be considered—signal averaging.

Trigger delay. Triggering is the process of starting the data acquisition at a desired point. For example, to acquire a single-shot pulse, a waveform digitizer might be set to trigger when its input rises to a certain level. Often, the system controller sets an instrument up to acquire data, but the acquisition cannot begin until a trigger occurs (Fig. 6-4).

If the trigger conditions are not satisfied, the acquisition never completes, and the whole system waits. The lesson here is: make sure the instrument(s) get the necessary trigger to begin their acquisition, and account for the time between when the controller sets up the acquisition and when the trigger occurs. Also, remember that a trigger that occurs before the controller finishes setting up the instrument is of no value.

Some instruments, like the 7612D Programmable Digitizer, have a pre-trigger and/or post-trigger feature that allows the digitizer to start acquisition before or after the trigger. In this mode, the 7612D acquires pre-trigger data before it becomes triggerable. While it is acquiring this data, triggers are ignored. This pre-trigger delay must be added to the trigger delay when performance estimates are being made.

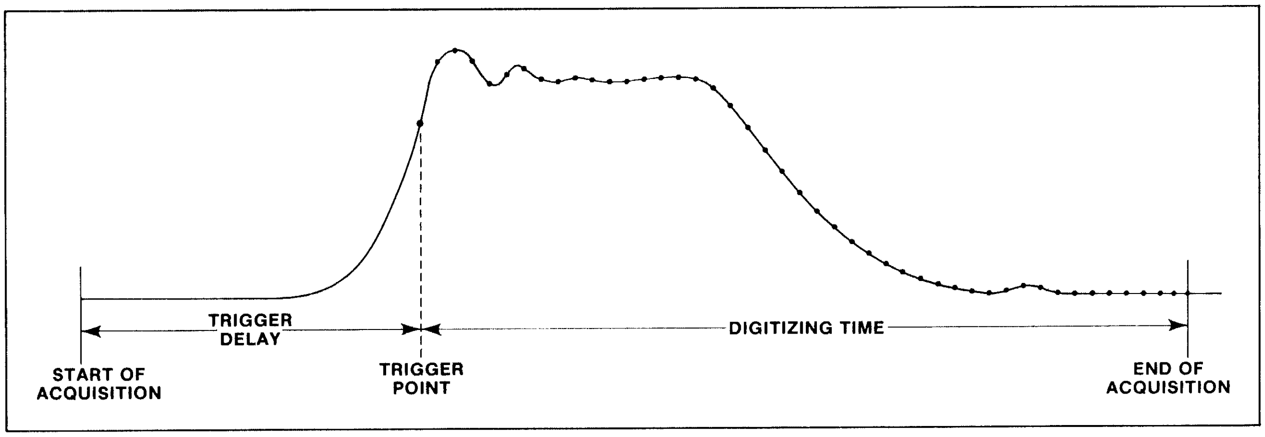


Fig. 6-4. *The total acquisition time is made up of trigger delay and digitizing time. In some digitizers, pre-trigger delay may also be added to this time.*

Digitizing time. When the instrument begins digitizing data, some time is required to complete the digitizing process. The time depends on the type of digitizer, the sample interval, the number of samples, and, in some cases, the sweep speed.

Digitizers fall into two basic classes—sequential and equivalent-time digitizers. Sequential digitizers acquire all data samples sequentially. The signal need not be repetitive because the complete waveform is acquired in a single pass.

Equivalent-time digitizers acquire their data in multiple passes or sweeps. If samples cannot be taken at a rate fast enough to acquire the waveform with sufficient resolution, the digitizer can acquire several cycles of a repetitive waveform. On each sweep, the samples are taken at a slightly different point on the waveform, so that after several sweeps enough samples are captured to accurately describe the input signal. Figure 6-5 illustrates the difference between sequential and equivalent-time digitizing.

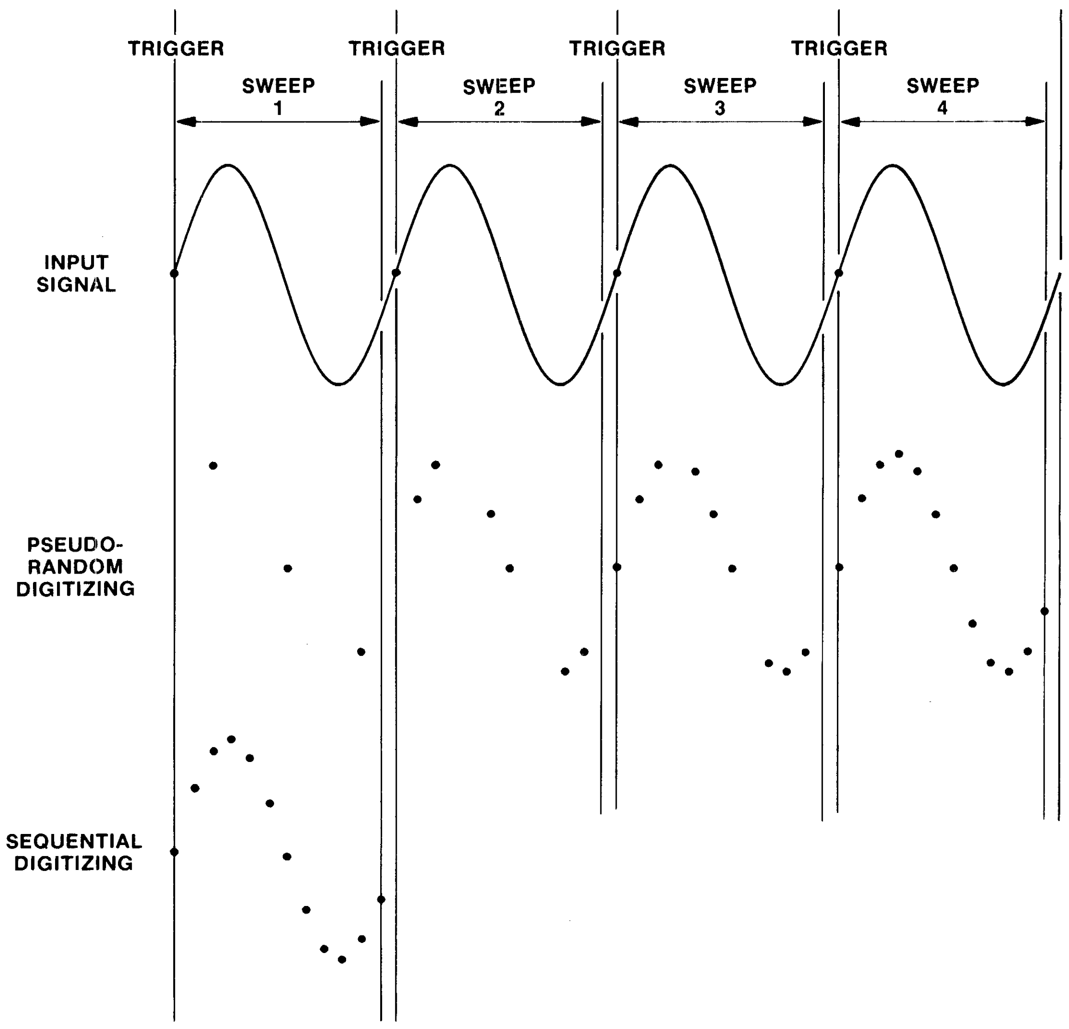


Fig. 6-5. *Equivalent-time digitizers sample a repetitive signal on several successive sweeps, gradually collecting enough samples to fully define the waveform. Sequential digitizers, on the other hand, acquire all the samples in a single pass. They must be capable of sampling the signal much faster than an equivalent-time digitizer.*

The acquisition time for a sequential digitizer is simple to calculate. It is just the number of samples times the sample interval. However, in an equivalent-time digitizer, samples are taken on several passes of a repetitive waveform. The acquisition is completed when a pre-defined number of points have been acquired. Thus, the acquisition time depends on the number of sweeps required to capture the points, the duty cycle of the waveform, and several other factors. The total digitizing time may be difficult to predict.

Signal averaging. Some digitizers offer the capability to repetitively acquire a waveform and average the data to remove random noise. This technique is very useful in many applications, but it is also time consuming. A complete acquisition is required for each average as well as time to compute the average. If an equivalent-time digitizer is used, the signal average may take quite a long time since every acquisition requires several sweeps and signal averaging requires multiple acquisitions.

Human Interaction

Sometimes human intervention is required to enter parameters, make adjustments to non-programmable instruments' settings, or set up tests. Human interaction time is an important consideration if the system will require input or other assistance from an operator. This interaction can often be minimized with careful system design and the use of fully programmable instruments. If execution speed is a critical factor in the system design, the amount of human interaction should be minimized.

Section 7 — Improving GPIB System Performance

Improving the performance of a GPIB system can be an elusive goal. The first step is to identify the components that affect the system performance and estimate how each component contributes to the overall performance. The previous section described each of these components and discussed techniques for estimating the time required for each. This section provides some hints and techniques to improve system performance.

Know Your Instruments

To write efficient programs for controlling GPIB instruments, a good understanding of the instruments is essential. It's important to know how they buffer and execute commands and how the commands interact. Know the data formats used to transmit and receive data. And, if several formats are available, know which ones the controller can transmit or receive fastest.

Also, know how the instrument acquires data. Does it respond to requests for waveform data while an acquisition is in progress? Does it signal the controller with an SRQ when data acquisition is complete?

Know the different conditions that can cause the instrument to assert SRQ. The programs you write will have to deal with any of these conditions that might occur during execution. Many instruments provide commands to disable specific interrupt sources, as well as the entire SRQ function.

The Operators or Programmers manuals should provide the specific information you need about the instruments. Read the programming and operation information carefully before beginning to write programs. A few minutes spent familiarizing yourself with the instruments can save hours of programming problems and frustrations, as well as produce a far more efficient system.

Choosing the Right I/O Statement

In many GPIB systems, data transfer takes a significant part of the total execution time. Significant performance improvements can be realized by carefully choosing the right I/O statements for each transfer and by carefully constructing the statement to minimize the amount of bus traffic.

PRINT. The PRINT statement provides a convenient means of transmitting ASCII data over the GPIB. It provides automatic addressing and unaddressing and data conversion and extremely flexible data formatting. Since the majority of GPIB instruments and peripherals communicate with ASCII, the PRINT statement is usually the best choice for sending device-dependent messages.

INPUT. The INPUT statement is the counterpart of PRINT for receiving data from the GPIB. It can receive data in a variety of formats and store it in string variables, numeric variables, or arrays. Like PRINT, the addressing and unaddressing functions are handled automatically. Most GPIB instruments transmit data and query responses in ASCII, so INPUT is usually the easiest way to receive the data.

WRITE. The WRITE statement is designed to transmit data on the GPIB in 4052A internal binary format. Since few GPIB instruments understand this format, it is not particularly useful for instruments. However, it can be used to write data to GPIB storage peripherals. Since no data conversion is necessary on input or output, the transfer is considerably faster than with other means. Addressing and unaddressing are handled automatically as with PRINT and INPUT.

READ. The READ statement is designed particularly to accept data in 4052A internal binary format. It is seldom useful for communicating with GPIB instruments, but can be used with GPIB storage peripherals. When data is written to a storage peripheral with WRITE, the READ statement can recover this data at high speed since no data conversion is necessary for internal binary format. Addressing and unaddressing is automatic.

WBYTE. The WBYTE statement is designed to provide low-level control of the GPIB for functions that cannot be implemented with the high-level statements. It gives you complete control over the GPIB data bus and the GPIB attention (ATN) and EOI lines. Any byte can be transmitted. However, WBYTE cannot transmit data from string variables—all data must be numeric constants, numeric variables, or numeric expressions that reduce to a value of -255 to +255.

WBYTE is useful for transmitting binary data (not internal 4052A binary format), setting up peripheral-to-peripheral transfers, and sending interface messages. The added complexity of using WBYTE can be offset by considerable speed advantages of transferring data in binary rather than ASCII. This subject is discussed in detail under the heading **ASCII vs. Binary—Simplicity vs. Speed**, later in this section.

With WBYTE, addressing and unaddressing are not done automatically; you use the WBYTE statement to address or unaddress an instrument. Addresses that you specify in a WBYTE statement must be absolute addresses.

RBYTE. The RBYTE (Read Byte) statement is the low-level counterpart of WBYTE. It accepts data from addressed talkers and assigns it to numeric variables. RBYTE can only store data it receives in numeric variables, and the largest value it can receive is 255 (the largest value that can be sent in a single byte). No addressing is required with RBYTE, because it assumes that a device has already been assigned to talk with WBYTE.

Minimizing Bus Traffic in PRINT

The PRINT statement provides an almost infinite variety of output formats. This variety allows you to format the output data almost any way an instrument needs it. But, it also means that a slight difference in the format of the parameter list can make a big difference in the number of bytes added for formatting a message and, as a result, the transfer rate.

The most striking example is the use of the semicolon or comma delimiter between variables in the PRINT statement. If a semicolon is specified between variables or constants in the parameter list, samples (variables or array elements) are delimited by one space. Thus, one extra byte per sample is transmitted. If a comma is specified between variables, samples are formatted into an 18-character field by adding spaces. For a typical four-digit data sample, 14 spaces are added, for a total of 18 bytes transmitted with each sample.

When array variables are transmitted, using the semicolon delimiter causes the elements to be separated by a single space. For example:

```
10 DIM A(100)
   :
50 PRINT @4:A;
60 PRINT @4:A
```

Transmitting four-digit data samples, the PRINT statement in line 50 transmits 5 bytes per sample compared to 18 bytes per sample if transmitted by line 60. The result is that the statement in line 50 executes about 3.6 times faster!

If an instrument or peripheral requires a delimiter other than spaces between array elements, the PRINT USING statement can insert just about any delimiter or series of delimiters between the array elements.

A simple example is an instrument that requires each data sample delimited by a carriage return.

```
10 DIM A(100)
   :
50 PRINT @4:USING 100:A
   :
100 IMAGE 100(5D,/)
```

The PRINT statement in line 50 tells the 4052A to transmit array A using the format specified in the IMAGE statement of line 100. The IMAGE statement specifies that 100 five-digit numbers will be printed. All samples are formatted into five-character fields by adding spaces. The slash indicates that each sample should be followed by a carriage return.

Any character can be used as a delimiter by replacing the slash in the above example with the desired character placed in quotes. For example, if a comma is required, the following statement could be used:

```
100 IMAGE 100(5D,",")
```

Be sure you know the format of the data when setting up the IMAGE statement. Attempting to transmit a value with more digits than specified in the IMAGE statement causes an error. Specifying too large a data field slows the transmission down because the samples are padded with spaces to fill the field.

The program in Fig. 7-1 generates an array of numbers from 1 to 100 and prints each value, delimited by commas, on the 4052A screen. Using a carefully constructed IMAGE statement, extra "padding" bytes are eliminated, increasing the data transfer rate. Though the values are only printed on the screen in this example, the same technique applies to transmitting data over the GPIB.

The first four lines of the program generate the array. Then, line 50 prints the array on the graphic system display with each element delimited by a comma. The "FD" field operator in the IMAGE statement (line 60) prints the values in fields just large enough for the number with no added spaces. For the first nine samples (1-9), a single digit field is printed. The next 90 samples (10-99) are printed in two-digit fields, and the last sample (100) is printed in a three-digit field. All samples except the last one are followed by a comma.

```
10 DIM A(100)
20 FOR I=1 TO 100
30   A(I)=I
40 NEXT I
50 PRINT USING 60:A
60 IMAGE 99(FD,","),FD

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,
40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,
58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,
76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,
94,95,96,97,98,99,100
```

Fig. 7-1. *A simple program illustrating the use of the PRINT USING statement. The output printed on the screen is also shown.*

Synchronizing the Controller and Instruments

When a GPIB instrument is operating under program control, two programs are running, not just one. One program is running in the controller, and another in the microprocessor system in the instrument. It is important that these two programs be synchronized. Fortunately, synchronizing the programs is easy with Tektronix instruments.

Program execution in a Tektronix programmable instrument is controlled by the messages it receives from the GPIB. Messages go into an input buffer as they are received. Message processing begins when the instrument receives a message terminator or when the instrument's input buffer fills up. While the instrument is busy processing the message, it sets bit 5 (the busy bit) in its status byte and asserts $\overline{\text{NRFD}}$ (Not Ready For Data) to hold off further input.

While $\overline{\text{NRFD}}$ is asserted, the 4052A cannot send any more messages to the instrument or get any output from it. To illustrate, the following program repeatedly sends the FREQ command to a 492P Spectrum Analyzer incrementing the value on each pass through the loop.

```
100 FOR I=1 to 10
110   PRINT @1:"FREQ ";I;"GHZ"
120 NEXT I
```

This loop is executed much slower than it would if the message in line 110 were simply printed on the display. That's because the 492P will not accept the message until the message from the last loop is processed. Thus, the 4052A waits at line 110 on each loop for the 492P to finish execution of the last message.

This also works for input (for most Tektronix instruments). The program below increments the center frequency for a 492P just like the last one did, but it also asks for the frequency range at the end of each message.

```
100 FOR I=1 TO 10
110   PRINT @1:"FREQ ";I;"GHZ;FRQRNG?"
120   INPUT Freq(I)
130 NEXT I
```

On each pass through the loop, the FREQ command is sent and the frequency range is queried. Immediately after the message is sent in line 110, program control advanced to line 120. The talk address for the 492P is sent, but it does not begin talking until the message in line 110 is completed and the frequency range query response is ready.

Interrupts Boost Performance

Using the operation complete interrupt. A large part of a system's execution time can be consumed by waiting for asynchronous events, such as the completion of an acquisition. If the system controller is tied up waiting for these events, the performance of the system can be seriously degraded. The SRQ interrupt feature of many GPIB instruments can be used to minimize this waiting time and therefore, improve the performance of the system.

Consider, for example, a system consisting of a 4052A and one or more 7612D Programmable Digitizers. A simple program to acquire a waveform from the 7612D is shown in Fig. 7-2.

```
10 DIM Data(512)
20 PRINT @1,0:"ARM A;READ A"
30 WBYTE @65,96:
40 RBYTE Header,Bytecount_hi,Bytecount_lo,Data,Checksum
50 WBYTE @95:
```

Fig. 7-2. *A simple program to acquire a waveform from a 7612D Programmable Digitizer.*

This program sends the 7612D an ARM command to prepare channel A to acquire data. When a trigger is received, the acquisition begins. Until the acquisition is complete, channel A data cannot be read.

Immediately following the ARM command is a READ command. Since the channel A data cannot be read yet, the 7612D buffers the command and waits for the acquisition to complete. Line 30 assigns the 7612D to talk, but since data cannot be read from channel A yet, the 7612D holds the bus in a waiting state until the acquisition is complete. As a result, program execution is held at line 40, waiting for the 7612D to begin transmitting its data.

During this waiting period, no device-dependent message traffic can occur and no other processing takes place—the time is essentially wasted. If the 4052A could be freed to perform other tasks during this period, the system performance could be significantly improved. The problem is how to signal the 4052A when the acquisition is complete so that the process of reading the data can be initiated.

That's where SRQ interrupts come in. The 7612D, like most other acquisition instruments from Tektronix, have an Operation Complete or Waveform Readable interrupt feature that allows them to generate an SRQ when data becomes available for the controller. The program shown in Fig. 7-3 is the one shown in Fig. 7-2 with statements added to use the Waveform Readable interrupt feature of the 7612D.

```
10  DIM Data(512)
20  ON SRQ THEN 2000
30  PRINT @1,0:"WRI ON;ARM A"
40  REM *** PROGRAM LINES TO PERFORM OTHER ***
50  REM *** SYSTEM TASKS ARE ADDED HERE     ***
      :
      :
2000 POLL Device,Status;1,0
2010 IF Status>127 AND INT(Status/4) MOD 2=1 THEN
2020    PRINT @1,0:"READ A"
2030    WBYTE @65,96:
2040    RBYTE Header,Bytecount_hi,Bytecount_lo,Data,Checksum
2050    WBYTE @95:
2060 ENDIF
2070 RETURN
```

Fig. 7-3. *Adding a few lines to the program in Fig. 7-2 takes advantage of the waveform readable interrupt feature of the 7612D.*

This program begins by setting up an ON ... THEN statement for the SRQ interrupt. Then, line 30 sends the WRI ON and ARM A commands to the 7612D. These commands enable the WRI (Waveform Readable Interrupt) and arm channel A for acquisition. At this point, the 4052A is free to perform other tasks, such as process data from the last acquisition or service other instruments.

When the 7612D completes the acquisition, it asserts SRQ. This interrupts the currently executing routine and causes control to be passed to the routine starting at line 2000. Line 2000 polls the 7612D (other instruments could be added to the POLL list, but are omitted for clarity). Line 2010 tests the state of bits 3 and 8. If they are both set, a Waveform Readable interrupt for channel A has occurred and the binary data is read in lines 2020 through 2050. Otherwise, control is passed directly to the RETURN statement and the interrupt is ignored.

The performance improvement gained with this technique is a function of the instrument's acquisition time. For example, if the 7612D is acquiring a 2048-point waveform at one millisecond sampling interval, the total acquisition time is over 2 seconds (excluding pre-trigger delay time). By using interrupts, this time can be used for processing. For faster sampling intervals or waveforms with fewer points, the gain will be less dramatic.

Prioritizing serial poll response. When a system consists of several instruments that can assert SRQ, it is possible that more than one instrument will assert SRQ at the same time. It may be more important to respond to an SRQ from one instrument or group of instruments than from others in the system. For example, a digital voltmeter might take a new sample every 300 milliseconds and assert SRQ each time it does. Since the sample will be overwritten by a new one every 300 milliseconds, it's important to respond to the DVM's interrupt and read the data before the next sample is taken.

If the system also contains other devices whose interrupts don't need such timely response, the response to these interrupt must somehow be prioritized so that the most important devices are serviced first. Prioritizing the serial poll response is simple—just list the devices' numbers in the POLL statement in priority order from highest to lowest priority. If the DVM in the previous example is the highest priority device in the system, it should be listed as the first device in the POLL address list, followed by the next most important device, and so on until all devices that can assert SRQ are listed.

For example, Fig. 7-4 shows an interrupt handling routine for a system with a digital voltmeter (DVM), a function generator, and a magnetic tape drive.

When an SRQ occurs, program control is transferred to line 100. The POLL statement begins a serial poll process, starting with the first device in the list. Since the DVM's interrupt is the most important, it is checked first. If it is asserting SRQ, it is serviced first even if other devices are also asserting SRQ. If the DVM is not asserting SRQ, the next device in the list is polled, and so on until the device that is asserting SRQ is found.

Local Data Processing

Many GPIB instruments are capable of performing some on-board processing. This processing may be faster than sending data over the bus to the controller and performing the processing in the controller, especially if the processed data must be shipped back to the instrument for display or further processing.

A good example is the signal average function provided on many instruments. To signal average in the controller, each waveform must be sent to the 4052A. If a large number of waveforms are averaged, the time to transmit the waveforms can become quite substantial. If the same function is performed in the instrument, the data can be acquired and averaged and a single averaged waveform sent to the controller. In most cases, this process is considerably faster than performing the average in the 4052A.

Another example is illustrated in the two programs shown Fig. 7-5. Part **a** of the figure shows a program that acquires a spectrum from a 492P Spectrum Analyzer and finds its maximum value using the 4052R07 ROM Pack MAX function. The program in part **b** of the figure accomplishes the same function using the 492P FMAX and POINT commands.

```
10 PRINT @1:"CURVE?"
20 INPUT _curve
30 CALL "MAX",_curve,_max,_point
```

(a) *Finding the maximum value of the waveform in the 4052A.*

```
10 PRINT @1:" FMAX;POINT?"
20 INPUT _max,_point
```

(b) *Finding the maximum value of the waveform in the 492P.*

Fig. 7-5. *Two programs to find the maximum value of a digitized spectrum acquired by the 492P. The program in part **a** performs the operation in the 4052A, while the program in part **b** performs the same operation in the 492P.*

The program in part **a** executes considerably slower because it takes about 2 seconds to transmit 1024 points of ASCII data to the 4052A. (This time could be reduced to about 800 milliseconds if the data is transmitted in binary). In program **b** only a single point is transmitted since the 492P finds the maximum value itself and transmits it to the 4052A. As a result, this technique is much faster.

```
100 POLL Device,Status;4;1;3
110 GOSUB Device OF 130,200,300
120 RETURN
130 REM *** SRQ HANDLING FOR THE DVM ***
     :
190 RETURN
200 REM *** SRQ HANDLING FOR THE FUNCTION GEN. ***
     :
290 RETURN
300 REM *** SRQ HANDLING ROUTINE FOR THE MAG TAPE DRIVE ***
     :
390 RETURN
```

Fig. 7-4. *An interrupt handling routine for a GPIB system with a DVM, function generator, and a tape drive.*

ASCII vs. Binary—Simplicity vs. Speed

GPIB instruments and controllers usually transfer numeric data such as waveforms in one of two formats—ASCII-coded decimal numbers or binary numbers. Sometimes you have no choice which format to use because the instruments or controller require one format. Other instruments, like the 492P Programmable Spectrum Analyzer can transmit data in either format.

The question in this case is; Which format should I use? Each format has its advantages. ASCII data transmission is usually simpler. But, binary data transmission can be significantly faster because fewer bytes are transferred.

In the 4052A, ASCII data can be transmitted and received using simple PRINT and INPUT statements. Data storage format is also more flexible with ASCII. Numeric data may be stored in string variables or numeric variables. A wide variety of input terminators are also available to simplify breaking the data into manageable blocks.

However, ASCII data transmission is slow compared to binary transmission. In ASCII, a single numeric value is converted to a string of ASCII characters before transmission (Fig. 7-6). For example, the value 237 is converted to the ASCII code for "2", followed by the ASCII code for "3" and the ASCII code for "7". Usually, a delimiter character (such as space or comma) is appended before or after the string, for a total of four bytes.

Binary data transmission is considerably faster, since a one or two-byte binary equivalent of the number is transmitted. Any value between 0-255 can be transmitted in a single byte, and any value between 0-65,535 can be transmitted in two bytes. However, binary transfers are more complex to implement. Data is always stored in numeric variables and if the value is larger than 255, it requires two variables—one for each byte.

Though the low-level RBYTE and WBYTE statements in 4050 BASIC transfer data slower than the PRINT and INPUT statements, the number of bytes sent is drastically reduced, which usually

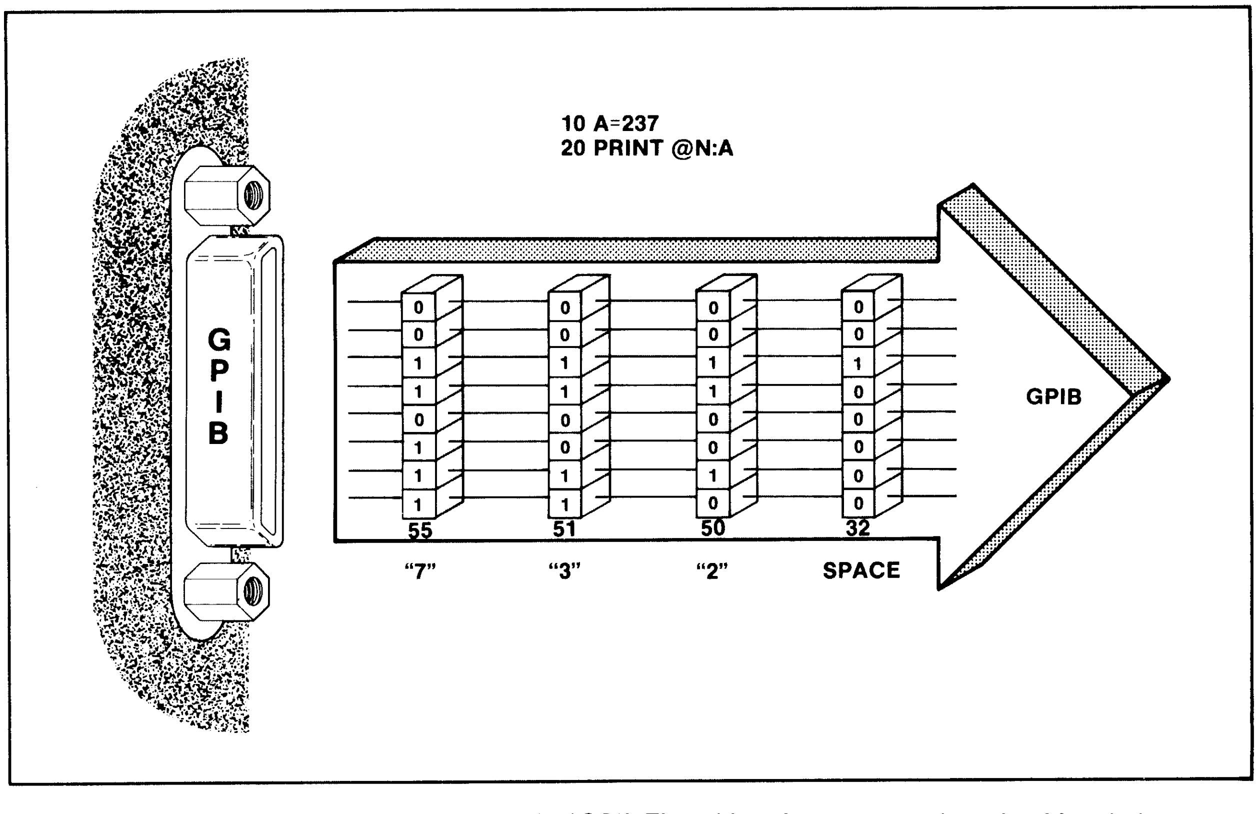

```
10 A=237
20 PRINT @N:A
```

Fig. 7-6. *Transmitting a numeric value in ASCII. The addressing sequence is omitted for clarity.*

more than compensates for the slower transfer rate. If the instrument you are using can not transmit/receive as fast as the 4052A INPUT/PRINT data burst, the full speed of the ASCII transfer can not be realized and the difference in speed between the binary transfer and the ASCII transfer will be even more pronounced.

To get an idea of the speed difference between ASCII and binary data transfers, refer to Table 7-1. This table shows some typical times to transfer a 1024-point waveform from the 7D20 to the 4052A in both ASCII and binary format. It's easy to see from this example why a slightly more complex program is a small price to pay when speed is important.

TABLE 7-1

WAVEFORM TRANSFER TIMES FOR 7D20/4052A SYSTEM

Binary transfer	800 ms
ASCII (input to string)	2.1 sec
ASCII (input to numeric array)	2.17 sec

In some cases, binary data may require some special processing to put it into a useable format. The time required to perform this processing must also be considered when choosing data transfer mode.

For example, some instruments send binary data in two bytes for each point. The controller has to assemble these two bytes into a single numeric value before the data can be processed. If performing this process requires a significant amount of time, it may be more efficient to transfer data in ASCII, even though the data transfer itself is slower.

Data Logging

In some cases, there simply isn't enough time to perform all the necessary processing on acquired data at the time it is acquired. A common solution to this problem is data logging, where data is captured and stored on a peripheral device at high speed. Then, after the acquisition is complete, the data is processed at a slower rate.

The WBYTE statement allows the 4052A to set up transfers between an acquisition instrument, for example, and a peripheral storage device. The process of setting up such a transfer is described in **Transfers Among GPIB Devices** in Section 4.

Section 8 — The GPIB ROM Pack

Introduction

This section describes the GPIB Enhancement ROM pack routines as they pertain to the GPIB programmer. The GPIB Enhancement ROM Pack (4052R14 Option 1A) is a useful tool for the GPIB programmer. It allows the following operations to be programmed more easily and executed more quickly:

- Binary transfers
- Sending multiline interface messages
- Bit-level operations (set, clear, test)
- Tape operations

In addition, the GPIB ROM pack allows you to do parallel polling and error handling operations that are not possible without the ROM pack. Table 8-1 lists the GPIB ROM pack routines along with a brief description of each.

TABLE 8-1

GPIB ROM PACK ROUTINES

ARSIZE	Returns the currently dimensioned size of an array.
ASKERR	Returns information that identifies the last error that was trapped.
BININ	Receives binary block data on the GPIB.
BINOUT	Sends binary block data over the GPIB.
CLRREP	Sets the error repetition counter to zero.
DCL	Sends the Device Clear interface message.
DECHEX	Converts decimal value to hexadecimal in ASCII representation.
ERRHLP	Returns information about Tektronix Standard Codes and Formats error codes.
ERRLIST	Enables you to specify which errors are to be trapped.
GET	Sends the Group Execute Trigger interface message.
GTL	Sends the Go To Local interface message with or without addresses.
HEXDEC	Converts an ASCII hexadecimal to decimal.
LAST	Finds and returns information about the last file on the tape in the 4052A internal tape drive.
LISTEN	Sends the listen addresses of the specified devices over the GPIB.
LLO	Sends the Local Lockout interface message.
LOCS	Puts all devices on the GPIB in local state by unasserting REN.
NEWTAP	Finds the present file location and returns the tape cartridge status.
PPD	Unconfigures specified devices for parallel polls.
PPE	Configures specified devices for parallel polls.
PPOLL	Performs parallel poll.
PPU	Sends the Parallel Poll Unconfigure interface message.
PRISTR	Outputs a string over the GPIB without carriage return or EOI.
RBIN	Receives binary data from the GPIB.
RETRY	Does a RETURN to the statement where the error occurred.
RWLS	Puts specified devices in Remote With Lockout State.
SDC	Sends the Selected Device Clear interface message.
SRQOFF	Causes SRQ interrupts to be ignored.
SRQON	A NOP in the 4052R14 option 1A ROM pack. Provides compatibility with software written for previous versions of the 4052R14 ROM pack.
STBHLP	Returns information about status byte codes.
TALK	Sends the talk addresses of the specified devices over the GPIB.
TAPEIN	Reads an entire ASCII file into a string variable.
TAPEAPP	Reads an entire ASCII file and appends the data to the end of a string variable.
THEADER	Finds the beginning of the current file and returns the file header or file number.
TNAME	Assigns a name to a file on the magnetic tape.
TRIM	Changes the current length of a string variable.
UNDEF	Determines whether a variable is defined or undefined.
UNL	Sends the Unlisten interface message.
UNT	Sends the Untalk interface message.
VARCLR	Clears specified bits in a target variable.
VARSET	Sets specified bits in a target variable.
VARTST	Tests specified bits in a variable.
VLIST	Returns all current variables and their values and types.
WBIN	Sends binary data.

Binary Transfers

The GPIB ROM pack provides four routines to do binary transfers:

- BININ—Receives data in binary block format.
- BINOUT—Outputs data in binary block format.
- RBIN—Receives binary data.
- WBIN—Outputs binary data.

All four routines can handle data in one or two-byte format. Unpacked mode, the default, is for one-byte data. Packed mode is for two-byte data. If you are dealing with an instrument that transfers data in two-byte binary format, these ROM pack routines offer a substantial savings in programming time and execution speed. For example, the 390AD Programmable Digitizer sends binary waveform data in two-byte format. The following program uploads 2048 points of waveform data from the 390AD (in dual mode) to the 4052A in about 3.8 seconds:

```
10 DIM Waveform(2048)
20 PRINT @1:"READ ch1"
100 CALL "BININ","PACK",Waveform,Error;1
```

The BININ routine (instead of RBIN) is used because the 390AD transmits binary waveform data in block binary format. "PACK" specifies two-byte data.

The following program uploads 2048 points of waveform data from the 390AD to the 4052A without using any GPIB ROM pack calls. It takes about 4.5 seconds to execute.

```
10 DIM Waveform(2048),Wave_tmp(2048,2),Constant(2,1)
20 Constant(1,1)=256
30 Constant(2,1)=1
40 PRINT @1:"READ ch1"
100 WBYTE @65:
110 RBYTE Header,Bytecount_hi,Bytecount_lo,Wave_tmp,Checksum,Semicolon
120 WBYTE 295:
130 Waveform=Wave_tmp MPY Constant ! Do a matrix multiply.
```

So, in this case, one ROM call takes the place of six statements and reduces the execution time by about 15 percent. The real benefit (besides the simplicity) is the memory savings afforded by the ROM pack routine. Uploading the waveform, using the ROM pack routine, only requires storage for 2048 numbers (2048 numbers * 8 bytes/number + 18 bytes for symbol table entry) plus 72 bytes for the CALL statement—a total of 16,834 bytes. Uploading the waveform with the fast matrix method requires storage for 6144 numbers (6144 numbers * 8 bytes/number + 18 bytes for symbol table entry) plus 432 bytes for the six statements—a total of 49,602 bytes. As you see, the fast matrix method requires over 3 times as much memory space as the ROM pack routine. If the 390AD was in "Channel 1 Only" mode, you could not use the fast matrix method because the 390AD then has 4096 points to send.

An alternative to the fast matrix method is to process each number (2 bytes) as it is received. This takes about the same amount of memory as that required with the ROM pack routine, but it is much slower (about 3 times slower). So the binary transfer ROM pack routines offer increased speed, more efficient memory usage and programming simplicity.

If you want to do error checking, the ROM pack routines can save you even more time. The binary transfer ROM pack routines automatically check for errors and return an error code.

You may use string variables instead of numeric variables with the four binary transfer ROM calls. Storing waveform data in strings can save memory space. To store an unpacked 2048 point waveform in a numeric array takes 16,402 bytes (2048 numbers * 8 bytes/number + 18 bytes for symbol table entry). To store the same information in a string takes 4114 bytes (2 hexadecimal digits/byte * 2048 bytes + 18 bytes for symbol table entry). But processing data stored in strings takes longer.

The data in strings used by the binary transfer routines is hexadecimal digits in ASCII format. To help process data in ASCII hexadecimal format, the GPIB ROM pack contains two routines to make conversions. HEXDEC converts a string of hexadecimal digits (maximum of four) to 4052A internal numeric format. DECHEX does the reverse.

Sending Interface Messages

The GPIB ROM pack provides routines to make it easy to send the following interface messages:

- DCL—Device Clear
- GET—Group Execute Trigger - addressable
- GTL—Go To Local - addressable
- LLO—Local Lockout

- SDC—Selected Device Clear - addressable
- UNL—Unlisten
- UNT—Untalk

In addition, there are routines to:

- Make a device a talker (TALK) - addressable
- Make a device a listener (LISTEN) - addressable
- Put a device into Local State (LOCS)
- Put a device into Remote With Lockout State (RWLS) - addressable

The routines marked addressable have optional address parameters. Without addresses, of course, TALK and LISTEN do nothing and RWLS without an address is the same as LLO.

These routines are easier to program than their equivalent in WBYTE statements. You can refer to the interface messages by mnemonic names and you don't have to add offsets for the listen, talk, and secondary addresses. This also makes programs easier to read. The operations performed by the five ROM pack routines that are not addressable are slightly faster using the ROM pack routines instead of WBYTE. The operations performed by the six addressable ROM pack routines are faster using WBYTE. However, the speed gained by using the low-level WBYTE would only be significant in time-critical applications or where the operation is done repeatedly.

Table 8-2 lists some execution times for the ROM pack routines and for equivalent operations using WBYTE. Times given are minimum. As always, the slowest listener on the GPIB determines the speed of the transfer. (An exception is the LOCS operation. REN is unasserted for a period of time that is independent of devices on the GPIB.)

Binary Operations

Three routines are available in the GPIB ROM pack to make binary operations easier: VARSET, VARCLR, and VARTST. With them you can set, clear, or test one or more bits in a numeric variable or array. The value of the object to be operated upon must fit in two bytes (-32768 to 65535).

One of the most common binary operations is testing the state of a bit in the status byte obtained with a POLL statement. The following statements test bits 3 and 8 of a status byte returned by a 7612D Programmable Digitizer to see if a Waveform Readable interrupt for channel A has occurred.

```
100 Bits_3_8=132
110 CALL "VARTST",Status,Bits_3_8,Is_it_set
120 IF Is_it_set=Bits_3_8 THEN
     :
     .
```

The VARTST routine does a bitwise AND on the integral parts of the first two parameters (STATUS and BITS_3_8 in this example) and stores the result in the third parameter (IS_IT_SET). Afterwards,

TABLE 8-2
Some Execution Times for Sending Interface Messages with the ROM Pack Routines and for Equivalent Operations Using WBYTE

ROM pack call	Execution time	Equivalent operation without ROM pack	Execution time
CALL "DCL"	1.01 ms	WBYTE @20:	1.19 ms
CALL "LLO"	1.03 ms	WBYTE @17:	1.19 ms
CALL "UNT"	1.06 ms	WBYTE @95:	1.19 ms
CALL "UNL"	1.06 ms	WBYTE @63:	1.19 ms
CALL "TALK";1	1.75 ms	WBYTE @65:	1.19 ms
CALL "LISTEN";1	2.02 ms	WBYTE @33:	1.19 ms
CALL "LISTEN";1;2;3;4	4.24 ms	WBYTE @33,34,35,36:	2.43 ms
CALL "GET"	1.44 ms	WBYTE @8:	1.19 ms
CALL "GET";1;2;3;4	5.77 ms	WBYTE @33,34,35,36,8,95,63:	3.67 ms
CALL "GTL"	1.44 ms	WBYTE @1:	1.19 ms
CALL "GTL";1;2;3;4	5.77 ms	WBYTE @33,34,35,36,1,95,63:	3.67 ms

CALL "LOCS"	0.62 ms	CALL "RENOFF" }
		CALL "RENON" }
			0.94 ms

IS_IT_SET and BITS_3_8 are equal if and only if the corresponding bits in STATUS are set. If, on the other hand, you wanted to know if **any** of the specified bits were set, comparing IS_IT_SET to zero would suffice.

Tape Operations

There are six routines in the GPIB ROM pack to facilitate tape operations:

- LAST—Finds and returns information about the last file on the tape in the 4052A internal tape drive.
- NEWTAP—Finds the present file location and returns the tape cartridge status.
- TAPEIN—Reads an entire ASCII file into a string variable.
- TAPEAPP—Reads an entire ASCII file and appends the data to the end of a string variable.
- THEADER—Finds the beginning of the current file and returns the file header or file number.
- TNAME—Assigns a name to a file on the magnetic tape.

These are general utility routines whose use is not limited to the GPIB programmer. As with the other ROM pack routines, these routines offer a savings in programming time and execution speed.

Error Trapping

Four routines are included in the GPIB ROM pack to enable you to trap 4052A system errors:

- ERRLIST—Enables you to specify which errors are to be trapped.
- ASKERR—Returns information that identifies the error that occurred.
- CLRREP—Sets error repetition counter to zero.
- RETRY—Does a return to the statement where error occurred.

To trap an error, call the ERRLIST routine and specify the error number or numbers you want to trap. You must write an error handler and specify its starting statement number in an ON SIZE THEN statement. Then, when any error specified in the ERRLIST call occurs, the 4052A does an implicit GOSUB to the statement number specified in the ON SIZE THEN statement. Of course, if a SIZE error occurs, the same action will be taken. The error handler must take action appropriate to the error that occurred. "Appropriate" could mean anything from an attempt to fix the problem, to ignoring it, to halting execution.

The following program segment is part of an error handler for a program that reads raw data from a measurement device on the GPIB and stores the data on the 4052A internal magnetic tape.

```
100 ON SIZE THEN 5000
110 CALL "ERRLIST",56,57,69
     :
     :
5000 REM ERROR HANDLER
5010 CALL "ASKERR",Index,Rp_count
5020 IF Rp_count>2 THEN
5030   PRINT "Three strikes - You're out"
5040   STOP
5050 END IF
5060 IF Index=1 THEN
5070   PRINT "The magnetic tape is write-protected. Please rotate the"
5080   PRINT "write-protect cylinder, on the tape, so the arrow points"
5090   PRINT "away from SAFE."
5100   PRINT "Press RETURN when when you have replaced the tape.";
5110   INPUT Tmp$
5120   GO TO 5260
5130 END IF
5140 IF Index=2 THEN
5150   PRINT "Please insert a tape in the"
5160   PRINT "4052A tape slot and press RETURN";
5170   INPUT Tmp$
5180   GO TO 5260
5190 END IF
5200 IF Index=3 THEN
5210   PRINT "There has been an error on the GPIB. Please check that"
5220   PRINT "all cables are connected and the instruments are"
5230   PRINT "powered up. Press RETURN to continue.";
5240   INPUT Tmp$
5250 END IF
5260 CALL "RETRY"
```

Line 100 tells the 4052A that the error trap handler starts in statement 5000.

Line 110 tells the 4052A that we want to trap errors 56, 57, and 69.

Line 5000 is the beginning of the error handler.

Line 5010 asks the 4052A to identify the error that caused the error trap. The repetition count is returned in RP_COUNT.

Lines 5020-5050 check the repetition count and bail out if the same error has occurred three consecutive times. The value of the repetition count is always one less than the number of successive times the error has occurred.

Lines 5060-5130 handle error 56.

Lines 5140-5190 handle error 57.

Lines 5200-5250 handle error 69.

All three parts of the error handler converge to line 5260 where the RETRY routine transfers control to the statement where the error occurred.

Error Codes, Event Codes, and Status Bytes

The 4052R14 GPIB ROM pack provides two routines to help decipher error and event codes, and status bytes:

- ERRHPL—Returns information about Tektronix Standard Codes and Formats error codes.
- STBHLP—Returns information about system status byte codes.

These routines map error and event codes, and status bytes to English explanations, thus eliminating the need for the programming time and memory space to accomplish the same task. The codes and status bytes recognized by the routines are those defined in the Tektronix Standard Codes and Formats. STBHLP only recognizes system status bytes. Your 4052R14 ROM pack manual lists the error and event codes that ERRHLP recognizes.

Parallel Polling

Four routines provided in the GPIB ROM pack enable you to perform parallel polls on instruments that implement the PP1 or PP2 subset:

- PPD—Unconfigures selected instruments for parallel polls.
- PPE—Configures selected instruments for parallel polls.
- PPU—Unconfigures all instruments for parallel polls.
- PPOLL—Performs parallel poll.

The first three operations can be accomplished with the WBYTE statement but are facilitated by the ROM pack routines. The last operation (PPOLL) can only be accomplished with the ROM pack because the 4052A must read a byte from the GPIB while asserting ATN and EOI.

Parallel polling is faster than serial polling and, if there are a large number of devices on the GPIB, can offer significant savings in polling time.

Before you can do a parallel poll, the devices you want to poll must be configured for parallel polling. When you configure a device for parallel polls, you tell the device to participate in upcoming parallel polls, on which data line (1 to 8) to respond, and what sense (logical 0 or 1) to use. The following ROM call configures devices at addresses 1 and 2 to use data line 1 and negative logic (0 is true):

```
CALL "PPE",0,1;1;2
```

Now, whenever you do a parallel poll, devices at address 1 and 2 will participate (along with any other configured devices). The following ROM call does the parallel poll:

```
CALL "PPOLL",P_status
```

It takes about 2.25 milliseconds to do a parallel poll. Unlike other GPIB operations, the speed of the parallel poll is not regulated by the devices on the bus. The participating devices must set the state of their assigned data lines within 200 nanoseconds. The amount of time it takes to read the data lines and unassert ATN and EOI is dependent solely upon the controller.

Miscellaneous Routines

There are nine miscellaneous utility routines in the GPIB ROM pack:

- ARSIZE—Returns the current dimensions of an array.
- DECHEX—Converts decimal value to hexadecimal in ASCII representation.
- HEXDEC—Converts an ASCII hexadecimal to decimal.
- PRISTR—Outputs a string over the GPIB without carriage return or EOI.
- SRQOFF—Causes SRQ interrupts to be ignored.
- SRQON—A NOP in the option 1A ROM pack. Provides compatibility with software written for previous versions of the 4052R14 ROM pack.
- TRIM—Changes the current length of a string variable.
- UNDEF—Determines whether a variable is defined or undefined.
- VLIST—Returns all current variables and their values and types.

The DECHEX and HEXDEC routines are useful with the string versions of BININ, BINOUT, RBIN, and WBIN. See the discussion of Binary Transfers in this section for more information.

The PRISTR routine is useful, in conjunction with WBYTE, for sending strings. With WBYTE, you can not send strings directly.

The functions of ARSIZE and UNDEF can also be performed by the more general UBOUND statement. The SRQOFF routine performs the same function as the OFF SRQ statement.

Appendix A — Subsets Describe GPIB Interface Functions

The IEEE 488 standard allows a designer a great deal of flexibility in implementing an instrument's GPIB interface. The functional capability of the interface is divided into ten interface functions. From this list of functions, a designer may choose to implement all, part, or none of each function, as defined by the interface subsets listed in the standard.

The standard also specifies a shorthand way of describing which optional functions are implemented in a device's interface. Each function is assigned a mnemonic (e.g., AH for Acceptor Handshake, DT for Device Trigger). A number appended to the end of the mnemonic indicates how many, if any, of the optional features of that function are implemented. Zero indicates that the function is not implemented at all. SH0, for instance, means that no source handshake capability is implemented in the interface. SH1 means that full source handshake capability is implemented.

Several of the interface functions have only two options—full capability or no capability. Others, such as the talker and controller functions, offer many options. Table 1 shows a summary of the interface subsets with a brief explanation of each function. The controller function is not included in the chart since it contains an extensive list of options. Refer to the IEEE 488 Standard for information on controller interface subsets.

It's important to remember that the interface subsets describe the repertoire of the interface only. They don't say anything about the programmable functions of the instrument. But, they're still important, because the programmable features of the instrument can't be used to full advantage without the appropriate interface functions. For example, if an instrument sends data over the bus, the talker and source handshake functions must be implemented in its interface. However, the designer may choose from one of nine levels of basic talker capability or nine levels of extended talker capability. The choice is based on which of the optional functions are required. If the instrument implements the T7 talker subset, it will have basic talker capability with talk-only mode, no serial poll capability and it will not remain a talker when addressed to listen.

The key is knowing which interface subsets your application requires. The interface subsets are usually listed in the specifications for GPIB instruments. Some instruments even print the subsets on the panel near the GPIB connector.

SOURCE HANDSHAKE		SH0	SH1	
Full capability	Allows a device to generate the handshake cycle for transmitting data		X	
No capability		X		

ACCEPTOR HANDSHAKE		AH0	AH1	
Full capability	Allows a device to generate the handshake for receiving data		X	
No capability		X		

TALKER (EXTENDED TALKER)*		T0 (TE0)	T1 (TE1)	T2 (TE2)	T3 (TE3)	T4 (TE4)	T5 (TE5)	T6 (TE6)	T7 (TE7)	T8 (TE8)
Basic Talker (Basic Extended Talker)	Allows an instrument to transmit data		X	X	X	X	X	X	X	X
Talk Only Mode	Allows an instrument to transmit data without a controller on the bus		X		X		X		X	
Unaddressed If My Listen Address (MLA)	Prevents an instrument from being a talker and a listener at the same time						X	X	X	X
Serial Poll	Allows an instrument to send a status byte in response to a serial poll		X	X			X	X		
No capability		X								

LISTENER (EXTENDED LISTENER)*		L0 (LE0)	L1 (LE1)	L2 (LE2)	L3 (LE3)	L4 (LE4)	
Basic Listener (Basic Extended Listener)	Allows an instrument to receive data		X	X	X	X	
Listen Only Mode	Allows an instrument to receive data with a controller on the bus		X		X		
Unaddress if My Talk Address (MTA)	Prevents an instrument from being a talker and a listener at the same time				X	X	
No capability		X					

SERVICE REQUEST		SR0	SR1	
Full capability	Allows an instrument to request service from the controller with the SRQ line		X	
No capability		X		

REMOTE-LOCAL		RL0	RL1	RL2	
Basic Remote-Local	Allows the instrument to switch between manual (local) control and programmable (remote) operation		X	X	
Local Lock-Out	Allows the return to local function to be disabled		X		
No capability		X			

PARALLEL POLL		PP0	PP1	PP2	
Basic Parallel Poll	Allows an instrument to report a single status bit to the controller on one of the data lines (DI01-DI08)		X	X	
Remote configuration	Allows the instrument to be configured for parallel poll by the controller		X		
No capability		X			

DEVICE CLEAR		DC0	DC1	DC2	
Basic Device Clear	Allows all instruments on the bus to be initialized to a predefined state cleared		X	X	
Selective Device Clear	Allows individual instruments to be cleared selectively		X		
No capability		X			

DEVICE TRIGGER		DT0	DT1	
Full capability	Allows an instrument or group of instruments to be triggered or some action started upon receipt of the Group Execute Trigger (GET) message		X	
No capability		X		

** Extended talkers and listeners use secondary addresses; other talkers and listeners do not.*

Appendix B — Glossary

absolute address—A talk address, listen address or secondary address. All absolute addresses are in the range 32 to 126. Absolute addresses are used with low-level WBYTE and RBYTE statements.

address—A numeric code that represents a unique device. The IEEE 488 standard defines three types of GPIB addresses; talk addresses, listen addresses and secondary addresses. Listen addresses are in the range 32-62. Talk addresses are in the range 64-94. Secondary addresses are in the range 96-126. Primary addresses 0-30 are used in GPIB I/O statements (except WBYTE and RBYTE) and are automatically mapped to the correct talk or listen address. For example, **PRINT @1:A$** refers to primary address 1 and the 4052A automatically converts primary address 1 to listen address 33.

AND—A binary Boolean algebraic operation that yields a value of true or "1" when both values are true or "1".

argument—A value or parameter included in a command. In a CALL statement, the parameters passed to the subprogram are called arguments. In a command for a Tektronix GPIB command, the arguments follow the command header.

array—A collection of numeric data items referenced by a single variable name. In 4052A BASIC, arrays may be one- or two-dimensional (i.e., organized as rows, or rows and columns).

ASCII—American Standard Code for Information Interchange. The ASCII code assigns a particular 7-bit code to all the alphanumeric characters, a standard set of punctuation characters, and a standard set of control characters (see control character).

ATN—The GPIB Attention line. The controller asserts ATN when it is transferring interface messages. When ATN is asserted, all devices must listen and bytes on the bus are interpreted as interface messages instead of device-dependent messages.

asynchronous—An operation that is not synchronized by a clock signal or other timing information. The GPIB is said to be an asynchronous bus because the process of sending messages is not timed by a clock signal—the transfer proceeds at the rate of the slowest device involved in the transfer.

BASIC—An acronym for Beginner's All-Purpose Symbolic Instruction Code. BASIC is a simple easy-to-learn computer programming language. 4052A BASIC is a highly enhanced version of standard BASIC that provides many extensions for instrument control and programming ease.

binary—A number system used by computers in which there are only two possible values for each place—1 (on) or 0 (off). Each place in the binary number system has a value of 2^n where **n** is the position of the place. Thus, the value of the least significant place is 1 (2^0), the value of the next place is 2 (2^1), the next is 4 (2^2), etc., so that decimal 9 in binary is 1001.

binary block—A data format defined by Tektronix Standard Codes and Formats. In this format, a block of binary data is transferred starting with the ASCII "%" character followed by two bytes that indicate the number of bytes in the block, followed by the data bytes, a checksum, an optionally, a terminator character.

Boolean—An algebra system developed by George Boole. This system uses logical operations, such as AND, NOT, OR, NOR, etc., instead of mathematical operations such as addition, subtraction, division, etc.

buffer—An area of memory used as temporary storage for data. In the 4052A, the I/O buffer is an area of memory where data is stored temporarily during an input or output operation.

byte—A group of binary bits, usually eight bits, that is operated on as a unit. A single ASCII character is stored in one byte.

call—To branch to or transfer control to a specified subprogram or ROM routine.

checksum—An error detection scheme used to check the validity of stored or transmitted data. The checksum is computed by adding the value of all the bytes involved. In Tektronix Standard Codes and Formats binary transfers, the checksum is a single-byte value that is the sum of all the bytes in the block including the byte count bytes but excluding the block header. If the sum is larger that can be contained in 8 bits, it is truncated to the least significant 8-bits.

controller—A computer whose major task is control. In a GPIB system, the controller is the device that manages the bus. It sends commands to set up data transfers and control bus operations.

DAV—Data Valid. A GPIB line that is asserted when the talker has a valid byte on the bus.

DCL—Device Clear. A universal GPIB message that tells all instruments on the bus to execute a device-specific "clear" function (see SDC).

device-dependent message—A message transferred on the GPIB with ATN unasserted. The message content and syntax is not specified by the IEEE 488 standard, it is determined by the instrument requirements. Device-dependent messages control instrument settings and parameters, in contrast to interface messages which control message traffic over the GPIB.

delimiter—A character or signal used to separate one data item from another or to terminate a set of data. For example, a comma may be used to separate one numeric value from another when transferring several ASCII numeric values. The EOI (End-Or-Identify) signal on the GPIB is often used as a delimiter to terminate message transfers.

end binary block—A data format defined by Tektronix Standard Codes and Formats for transferring binary data. The end binary block starts with the "@" character followed by the binary data values.

EOI—The End Or Identify signal line on the GPIB. EOI is used in Tektronix instruments as a message terminator. When the talker sends the last message byte it asserts EOI to indicate the end of the message. EOI is also used during the parallel poll process.

equivalent-time digitizing—A technique of digitizing where samples are taken over several repetitions of the input signal. On each repetition, the digitizer captures one or more samples, gradually building a complete acquisition from many repetitions.

error code—A code, usually a number, that identifies an error. Two kinds of error codes you will find in a 4052A controlled GPIB system are 4052A system errors and instrument errors. 4052A system error codes are reported on the 4052A display screen when an error occurs. 4052A system error codes may also be returned by the ERRLIST routine if you have a GPIB ROM pack. Instrument errors are returned by an instrument in response to an ERR? or EVENT? query.

error trapping—When an error occurs in a program, the computer's operating system normally takes control and reports an error message to the user. Error trapping allows a program to call a user-written error handler routine when an error occurs in lieu of the normal operating system error processing.

event code—A numeric code returned by a Tektronix GPIB instrument in response to an EVENT? or ERR? query. It's the same as an instrument error code except that it doesn't necessarily connote an error condition—some event codes denote normal conditions, such as power-up or operation complete.

function—A special type of 4050 BASIC statement that returns a single numeric.

GET—Group Execute Trigger. An addressed GPIB interface message. All addressed listeners that implement the device trigger function (DT1 subset) initiate an instrument-dependent trigger function when they receive this message. For example, a digitizer might be set to trigger its acquisition on receipt of GET.

handler—A special subprogram or subroutine that is called when a specified condition occurs. For example, an error handler routine may be called when an error occurs (see error trapping). Handlers may be written to process error conditions, interrupts such as SRQs from a GPIB device, or I/O device conditions, such as end-of-file on a tape drive.

handshake—The process of coordinating a data transfer from one device to another. Some form of signal or flag is exchanged between the communicating devices to insure the integrity of the data transfer. In GPIB data transfers, three control lines are used to control the transfer of each byte. These lines are called "handshake" lines.

hard copy—A paper copy or printout of information stored in a computer.

header—A character or group of characters that identifies the following command or data. In a Tek Standard Codes and Formats command, the header is the first word that identifies the command, such as

FREQ for a command to set the frequency. The first character in a binary block is also called a header, in that it identifies the data format the follows.

IFC—The GPIB Interface Clear signal line. The system controller can assert this line to reset all interfaces to a known state.

Interface—The connection of two devices, or the circuits and programs that allow two devices to communicate, such as a computer and a printer. Several standard interfaces have been defined, such as GPIB and RS-232C that allow a variety of computers and peripheral devices from different manufacturers (printers, terminals, etc.) to be connected. "Human interface" is the interaction between computers and operators.

Interface messages—A GPIB message that controls interface operations in contrast to device-dependent messages which control device functions. Interface messages may either be uniline messages (a single line asserted, such as REN, SRQ, EOI, etc.) or multiline messages (bytes sent on the data lines with ATN asserted). Multiline interface messages may either be universal messages, affecting all devices on the bus (such as DCL) or addressed messages, affecting only the addressed listeners (such as SDC).

Interface subsets—A shorthand way of defining the interface capabilities of a GPIB device. The functions of a GPIB interface are divided into ten basic functions. These functions are further divided into subsets that describe which optional parts of each function are implemented in the interface. Appendix A describes the interface subsets.

Interrupt—A condition that causes a program to temporarily suspend execution of the current program and begin executing another task. When the new task is complete, execution may return to the point the original task was suspended, or it may be transferred to another point.

Interrupt trapping—When an interrupt occurs, such as an SRQ from a GPIB device, the computer's operating system normally processes the interrupt. Interrupt trapping allows the user to write a handler (see handler) routine to process the interrupts instead of using the operating system's interrupt processing routine (see error trapping).

I/O—Shorthand for Input/Output.

keyword—Some words have special pre-defined meanings to a computer. These words cannot be used as variable names within a program. For example, the word PRINT always means "output data" to the 4052A. As a result, PRINT is not a valid variable name. These reserved words are called keywords.

listen address—The primary address of a GPIB device plus 32. The listen address is the address used by the controller to address a device to listen. The address switches on the instrument set the primary address, which determines both the listen address and talk address of the instrument (see talk address and primary address).

listen-only—A GPIB device that has been manually configured as a permanent listener. The device does not need to be addressed by the controller to listen. A listen-only device and a talk-only device can work together on the bus without a controller.

listener—A GPIB device that has been addressed to listen by the controller or that is set to the listen-only mode. There can be any number of listeners in a system at any time (up to the limit of the number of devices on the bus).

local variable—A variable that can only be referenced within the program segment where it is defined, in contrast to a global variable that can be referenced anywhere in a program.

message terminator—Indicates the end of a message. Tektronix instruments usually assert EOI to indicate the end of message. Some devices send the line-feed character as a terminator at the end of a message.

NDAC—Not Data Accepted. A GPIB handshake line asserted by a listener when it has NOT captured the byte currently on the bus. All listeners must release NDAC before the talker can change the data byte on the bus or release DAV.

NOP—No Operation. An operation that has no effect.

NRFD—Not Ready For Data. A GPIB handshake line asserted by a listener when it is not ready to accept another data byte. All listeners must release (unassert) NRFD before the talker may place another byte on the bus and assert DAV.

peripheral device number—A number that represents a device or a GPIB address. Peripheral device numbers 0-30 correspond to GPIB primary addresses 0-30. The peripheral devices (internal magnetic tape drive, keyboard, ROM slots, etc.) are assigned peripheral device numbers greater than 30.

primary address—A numeric code from 0-30 that uniquely identifies a particular device on a GPIB bus. This code is usually set by a switch or switches on the GPIB device. The primary address actually defines two addresses—a talk address used to address a device to send data on the bus, and a listen address used to address a device to receive data from the bus. The listen address is the primary address plus 32 and the talk address is the primary address plus 64 (see listen address and talk address). 4052A peripheral device numbers 0-30 correspond to GPIB primary addresses 0-30.

real-time clock—A clock that measures time in terms of actual clock time (seconds, minutes, etc.), in contrast to a clock which measures some relative units, such as execution cycles for a computer.

record—A group of related information. In a waveform digitizer, a record is the set of samples that defines a waveform. In a computer file, such as on the 4052A tape, a record is one set of information, often terminated by a carriage return or other special character.

record length—The number of elements in a record. In a waveform digitizer, the record length is the number of samples in the record. In computer file, the record length is the number of characters or bytes in each record.

RAM—Random Access Memory. This term refers to memory that is organized such that any memory location can be read at any time. Though read-only memory (ROM) is also random-access in nature, the term RAM is usually reserved for memory that can both be read as well as written to. RAM is used in the 4052A for storing BASIC programs and data.

REN—The Remote Enable GPIB line. The controller asserts this signal to allow devices on the bus to operate in remote control from the bus. When REN is not asserted, devices must return to local (front panel) control.

ROM—Read Only Memory. 4052A BASIC and the ROM routines are stored in integrated circuits that are programmed at the factory with the operating system program. This information can only be read by the computer; it cannot be modified or written.

sample—A number that represents the value of something. That something may be the magnitude of a signal (from a digitizer or a multimeter), a frequency (from a spectrum analyzer), a time interval (from a timer) or anything else that can be characterized by a number.

sampling interval—The period between samples in a digitizer or other sampling device, usually expressed in seconds.

sampling rate—The reciprocal of the sampling interval.

scalar—A single numeric value.

secondary address—A numeric code that is used on the GPIB in addition to the primary address to identify either a sub-function of an instrument or to indicate a command. For example, in the 7612D Programmable Digitizer, the secondary address indicates whether the mainframe or programmable plug-ins will be involved in a data transfer. In the 4924 Digital Cartridge Tape Recorder, secondary addresses are used as commands, such as rewind, find file, etc.

sequential digitizing—A technique of digitizing a waveform where samples are taken at fixed intervals along the input waveform. All samples are captured, in order, on a single input repetition (compare equivalent-time digitizing).

SDC—Selected Device Clear. An addressed GPIB interface message that tells the addressed device to execute a device-specific clear function. The SDC message is similar to the DCL message except that SDC affects only the addressed devices, where DCL affects all devices on the bus.

serial—Handling or transferring data one item at a time, in contrast to parallel, where several data items are handled or transferred at once.

serial poll—A protocol used on the GPIB to read the status of devices on the bus. The controller reads one status byte from each device polled. If several devices are polled, their status bytes are read one at a time.

SPE—Serial Poll Enable. A universal GPIB interface message that tells all devices on the bus to prepare to send their status bytes when they are addressed to talk.

SRQ—The GPIB Service Request line. A device on the bus asserts this line to request service from the controller. The controller usually conducts a serial poll or a parallel poll to determine which device is asserting SRQ and to read the device status.

status byte—A byte returned by a GPIB instrument during a serial poll that contains information about the state of the instrument.

subprogram—A program segment in 4052A BASIC that is bounded by a SUB and an END SUB statement is called a subprogram. The subprogram is given a name in the SUB statement and is called by that name using a CALL statement. A subprogram can have independent (local) variables that are defined only within that subprogram. In addition, parameters may be passed between the caller and subprogram through the CALL statement.

subroutine—A part of a 4052A BASIC program that is called with a GOSUB statement and is terminated with a RETURN statement. Subroutines, in contrast to subprograms, share all their variables with the context they are in. In addition, the GOSUB statement cannot pass parameters to the subroutine like a CALL statement can.

talk address—The primary address of a GPIB device plus 64. The talk address is the address used by the controller to address a device to talk. The address switches on the instrument set the primary address, which determines both the listen address and talk address of the instrument (see talk address and primary address).

talk-only—A GPIB device that has been manually configured as a permanent talker. The device does not need to be addressed by a controller. A talk-only device can work together with a listen-only device without a controller.

talker—A device on the GPIB that has been addressed to talk by the controller or is set to the talk-only mode. There can only be one talker in a system at any time, though there may be any number of listeners (see listener).

Tektronix Standard Codes and Formats—A standard published by Tektronix that defines the content and syntax of device-dependent messages for Tektronix GPIB instruments. The standard provides for consistent syntax across all instruments that conform to the standard.

Index