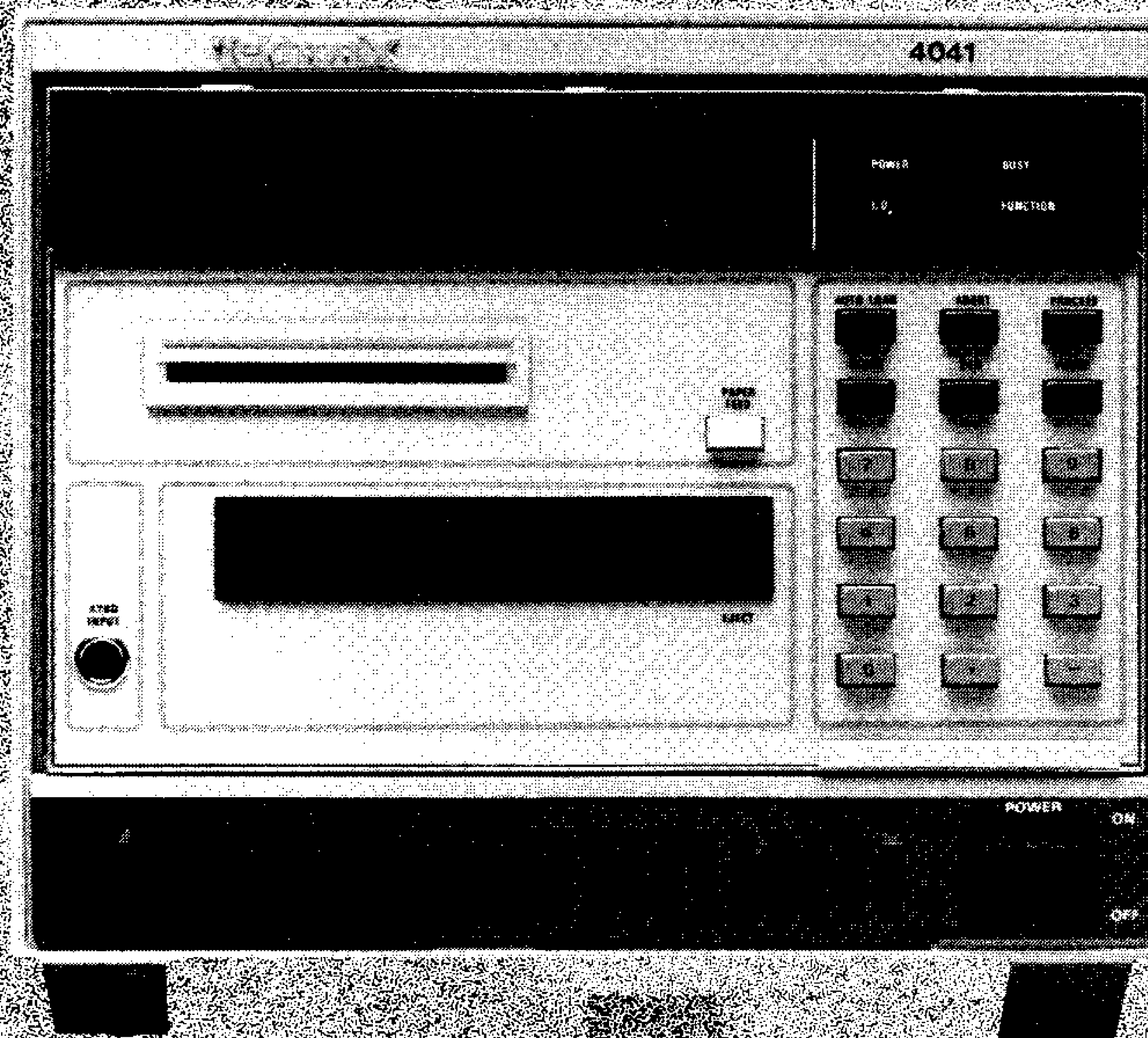
4041 GPIB Programming Guide



Tektronix®
COMMITTED TO EXCELLENCE

A binder is available from Tektronix for your GPIB Programming Guides and Instrument Interfacing Guides. Contact your local Tektronix 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
- Instrument Interfacing Guides (IIG)
- Other GPIB Programming Guides

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

The information presented in this programming guide is provided for instructional purposes only. Tektronix, Inc. does not warrant or represent in any way the accuracy or completeness of any program herein or its fitness for a user's particular purpose.

This Programming Guide was written by Mark D. Tilden and produced by the Applications Support Group.

Copyright © 1983 by Tektronix, Inc., Beaverton, Oregon.
Printed in the United States of America. All rights reserved.

070-4696-00

First Printing SEP 1983

Preface

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 Tektronix 4041 System Controller.

Section 1 provides a brief introduction to the 4041's GPIB capabilities. Section 2 discusses guidelines for choosing system components and configuring the system. A brief review of the fundamentals of 4041 BASIC is provided in Section 3. Then, Section 4 gets down to the specifics of GPIB system programming with the 4041. A generous supply of examples are included to help illustrate the concepts.

Section 5 deals with interrupt handling in 4041 BASIC, while Section 6 is devoted to techniques for processing and displaying the acquired data. Section 7 describes the factors that affect system performance and Section 8 provides some hints for improving the performance of the system.

The IEEE 488 interface subsets are described and a summary table of the subsets is provided in Appendix A. Appendix B briefly describes the Tektronix Standard Codes and Formats for waveform data. Finally, a glossary of common GPIB and system terms is provided in Appendix C.

This guide is not intended as a reference manual for 4041 BASIC. Only the features of 4041 BASIC that relate to GPIB system programming are described in detail. More complete reference information is available in the *4041 Programmer's Reference Manual*.

For complete information on the IEEE 488 bus (GPIB), refer to IEEE Standard 488-1978. Also, refer to the instrument manuals for specific information on programming GPIB instruments.

Section 1 — The 4041 as a GPIB Controller 1

Defining the System Controller's Job 1
Program Development 1
Controlling the System 2
Addressing instruments 3
Sending data and commands 3
Transmitting and receiving data 4
Handling interrupts 4
Processing the Data 4
Storing and Displaying the Data 4
GPIB Capability—More than GPIB Compatibility 4

Section 2 — Configuring a 4041 GPIB System 7

Defining the System's Job 7
Selecting System Components 7
Getting It Together 13
Setting the bus address 13
Setting the message terminator 14
Cabling the instruments 14
In Summary 15

Section 3 — Introduction to 4041 BASIC 17

4041 BASIC Enhancements 17
Input/Output in 4041 BASIC 19
I/O statements 19
I/O devices 19
The system console device 20
The system device 20
Stream specifications 20
Logical unit numbers 21
I/O parameters 21
Interrupt Handling 22
Debugging Programs 22
Breakpoints 22
Tracing a program 23

Section 4 — Programming a 4041 GPIB System 25

System Power-up 25
Power-up test 25
Power-up SRQ 25
4041GPIB Defaults 25
Default GPIB parameters 25
Default LUNs 25
GPIB Parameters 25
Physical parameters 25
Setting physical parameters 27
Logical parameters 27
Setting logical parameters 28

Talking to the Instruments 29
Device-dependent messages 29
Transferring a device-dependent message 29
PRINTing and ASCII message 30
Getting a response 32
Receiving long strings 32
Inputting numeric values 34
Transferring Waveforms 34
Tek Codes and Formats waveforms 34
Receiving ASCII data 35
Sending ASCII data 36
Receiving binary data 36
Block binary data input 37
Receiving end block binary data 37
Reading binary data into a string variable 38
Sending binary data 38
Special I/O Situations 38
Suppressing the EOM character on PRINT 38
Using alternate delimiters on INPUT 39
Proceed Mode 40
Proceed mode PRINT 40
Proceed mode INPUT 41
Is it done yet? 41
Using the BUFFER Clause 41
The I/O buffer 41
Defining an alternate I/O buffer 42
GETMEM and PUTMEM 43
Using GETMEM 43
GETMEM with binary waveforms 44
Using PUTMEM 45
Low-Level GPIB I/O 46
The SELECT statement 46
The WBYTE statement 46
WBYTE GPIB funcitons 46
Transfers among GPIB instruments 48
The RBYTE statement 49
The 4041 as a Talker/Listener 49
Talker/Listener programming 50
Knowing when to talk and when to listen 51
Storing and Retrieving Data on Tape 52
Reading and writing ASCII data 53
A special case—binary waveforms in string variables 53
Detecting the end of the file 53
The TYPE function 54
ITEM format files 54
Physical mode I/O 54

Section 5 — Processing Interrupts in 4041 BASIC 57

Interrupt Conditions 57
What is a Handler? 57
Calling a Handler 57

System Handlers 58
Enabling an Interrupts 58
Disabling an Interrupt 58
Exiting from a Handler 58
The ADVANCE statement 59
The BRANCH statement 59
The MONITOR statement 59
The RESUME statement 59
The RETRY statement 59
Nested Handlers 59
An Example Handler 60
GPIB Interrupts 60
SRQ Interrupts 61
Polling the devices 61
POLL statement forms 62
SELECTing a GPIB port for POLL 62
The SPE parameter and POLL 62
Status byte format 63
Processing the Status Byte 64
Differentiating system and device status 65
Reporting errors 65
Normal and abnormal condition status 66
Error logging 66
Using SRQ interrupt to control program flow 67
EOI Interrupts 67
IFC Interrupts 68
DCL Interrupts 68
MTA and MLA Interrupts 68
TCT Interrupts 70
ABORT Interrupts 70
Nested ABORT handlers 70
ERROR Interrupts 72
Setting up error handlers 72
ON ERROR statement order 72
Proceed mode I/O errors 72
User-defined errors 72
The ASK$("ERROR") function 72
OFF ERROR statements 73
IODONE Interrupts 73
KEY Interrupts 73
User-definable keys and the console device 73
Key queueing 73
The ASK("KEY") function 73

Section 6 — Processing and Displaying Data **75**

Array Processing 75
ROM Packs 76
Calling ROM pack routines 76
Graphing Data 77
4041R01 Graphics ROM 78
4041R02 Plotting ROM 78

Signal Processing 79
4041R03 Signal Processing ROM 80
Sample Program 81

Section 7 — Estimating 4041 GPIB System Performance 85

GPIB System Performance Factors 85
Instrument set-up time 85
Data acquisition time 86
Data transfer time 87
Data processing time 88
Human interaction time 88
Estimating Performance Factors 88
Estimating instrument set-up time 88
Estimating data acquisition time 89
Estimating data transfer time 91
An example—estimating data transfer rate for a PRINT statement 95
Estimating serial poll time 96
WBYTE and RBYTE timing 98
Estimating data processing time 98
Estimating human interaction time 99
Using the 4041 Real-Time Clock for Timing Measurements 99

Section 8 — Improving 4041 GPIB System Performance 101

Get to Know the System 101
Reducing Set-up Time 101
Reducing Data Acquisition Time 102
Reducing Data Transfer Time 104
Reducing Data Processing Time 108
Reducing Human Interaction Time 110

Appendix A — Subsets Describe Interface Functions 113

Appendix B — Tek Standard Codes and Formats Waveforms 117

Appendix C — Glossary 121

Index 127

Section 1 — The 4041 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 4041 System Controller, 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 it's 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 application software that controls the system. Since a large part of the total system cost is wrapped up in the software development, the controller should provide an environment conducive to good programming practice and that

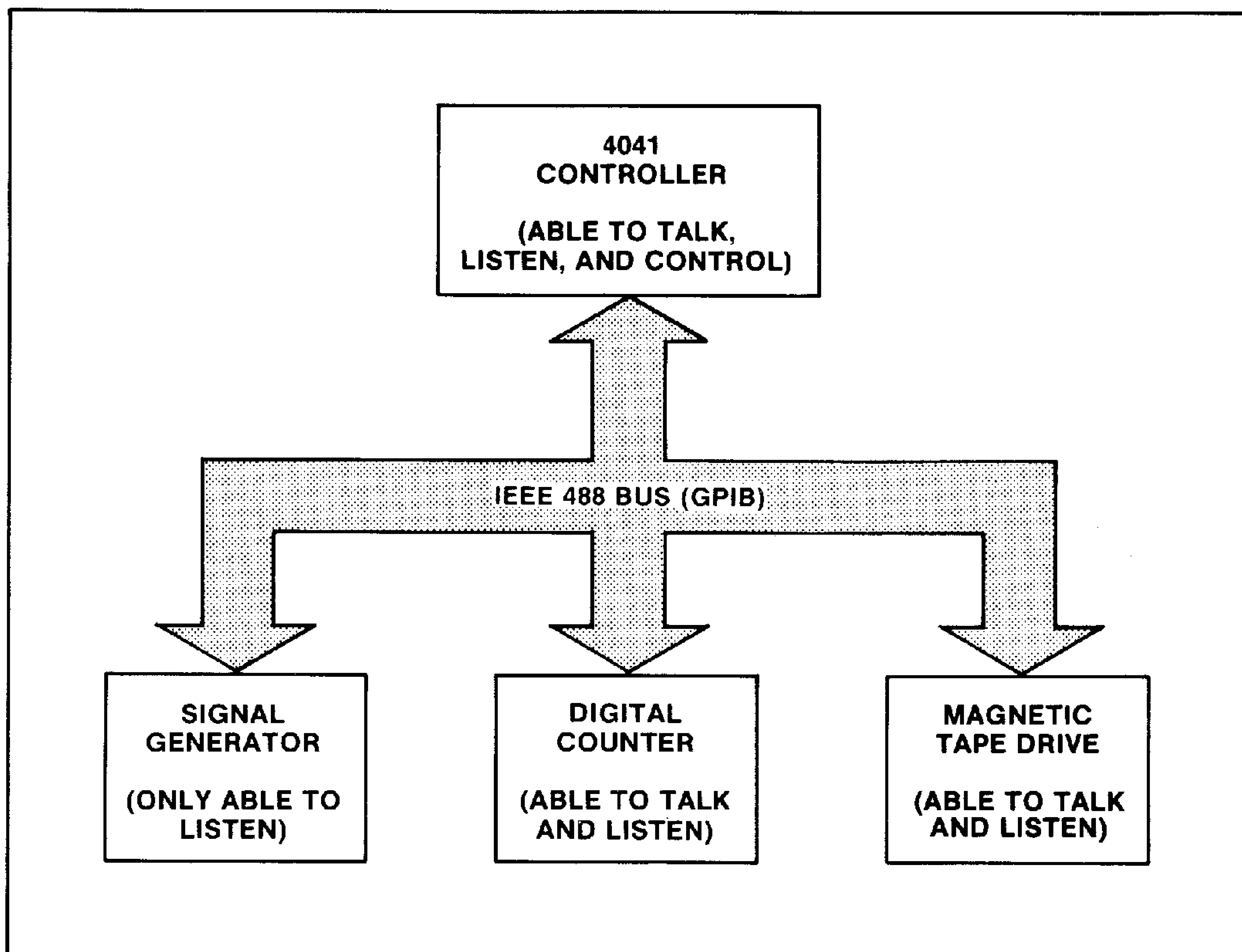


Fig. 1-1. *A typical GPIB system includes a controller and a variety of GPIB-interfaced devices with different capabilities.*

enhances programmer productivity. For example, full subprogram capability, long, descriptive variable names, local and global variables, and powerful debugging tools are all features that make software development and maintenance easier.

In addition, a full-size CRT terminal or display is helpful during program development and debugging. However, a CRT terminal is often not a good choice for interacting with operators that have minimal computer background. After the software is developed and running, the CRT terminal might be replaced with a simple keypad with user-definable keys. The simple keypad usually provides a better operator interface for system operators.

Finally, mass storage is important for storing programs and data. A standard, transportable media allows programs developed on one controller to be transferred to and run on many other similar controllers.

Controlling the System

Next, consider the task of instrument control. No matter how powerful the system components, 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 four functions:

Addressing instruments
Sending data and commands

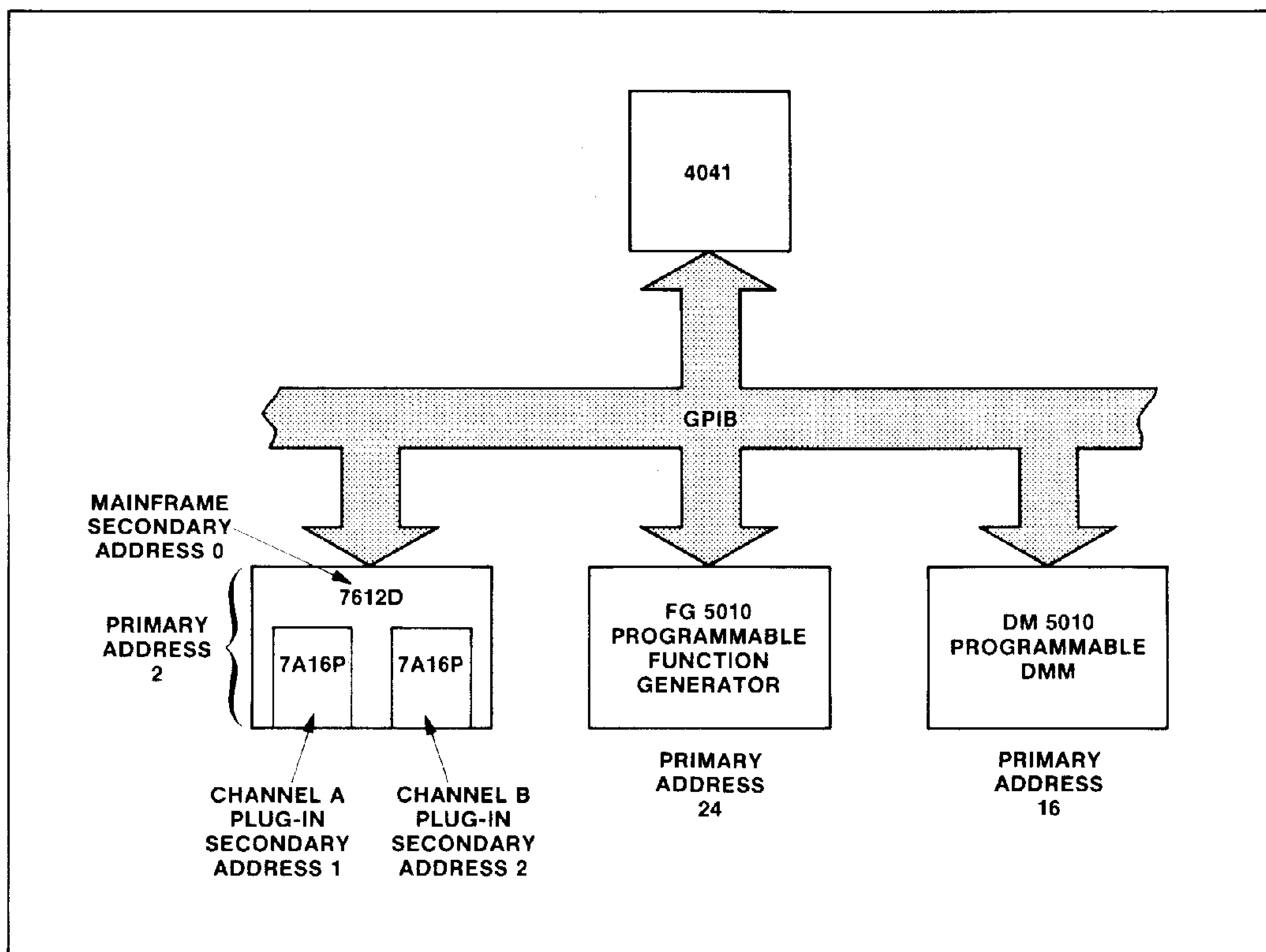


Fig. 1-2. *Each instrument on the bus is assigned a unique primary address. Secondary addresses are used in some instruments to select sub-sections or functions within the instrument.*

Transmitting and receiving data
Handling interrupts

Let's look at each of these functions individually.

Addressing instruments. The controller selects an instrument or set of instruments to be involved in a data transfer by "addressing" them. Every instrument is assigned a unique address in the range 0-30. This is the instrument's "primary" address. The controller uses the primary address to assign a device to talk or listen.

In addition, some instruments have another address or set of addresses called secondary addresses that select sub-sections or functions within the instrument. For example, the TEKTRONIX 7612D Programmable Digitizer has a secondary address that selects its mainframe and another secondary address for each programmable plug-in installed in the mainframe (Fig. 1-2). Thus, to address the right plug-in installed in a 7612D, the 7612D primary address is sent followed by the right plug-in's secondary address.

Sending data and commands. The controller sends two basic types of messages: device-dependent messages and interface messages (Fig. 1-3). Interface messages can be thought of as commands sent by the controller to direct interface operation. Device-dependent messages, on the other hand, consist of data and commands that control individual instruments on the bus. The controller uses the ATN (Attention) line on the GPIB to distinguish the two message types. When ATN is asserted, information on the bus is interpretted as interface messages. When ATN is not asserted, the messages are device-dependent.

The content and format of device-dependent 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.

Interface messages can be further divided into three types: Uni-line messages, Universal multi-line messages, and Addressed multi-line messages (Fig. 1-4). The IEEE 488 standard defines three bus control lines—REN (Remote Enable), ATN (Attention), and EOI (End Or Identify)—as interface messages in themselves. They are called uni-line messages. When the standard refers to the REN message, it simply means that the Remote Enable (REN) line is asserted.

The second and third type of interface messages are multi-line messages. These messages are sent by placing a byte on the bus with ATN asserted. Multi-line messages may either be universal commands, affecting all devices on the bus, or addressed commands affecting only the addressed instruments.

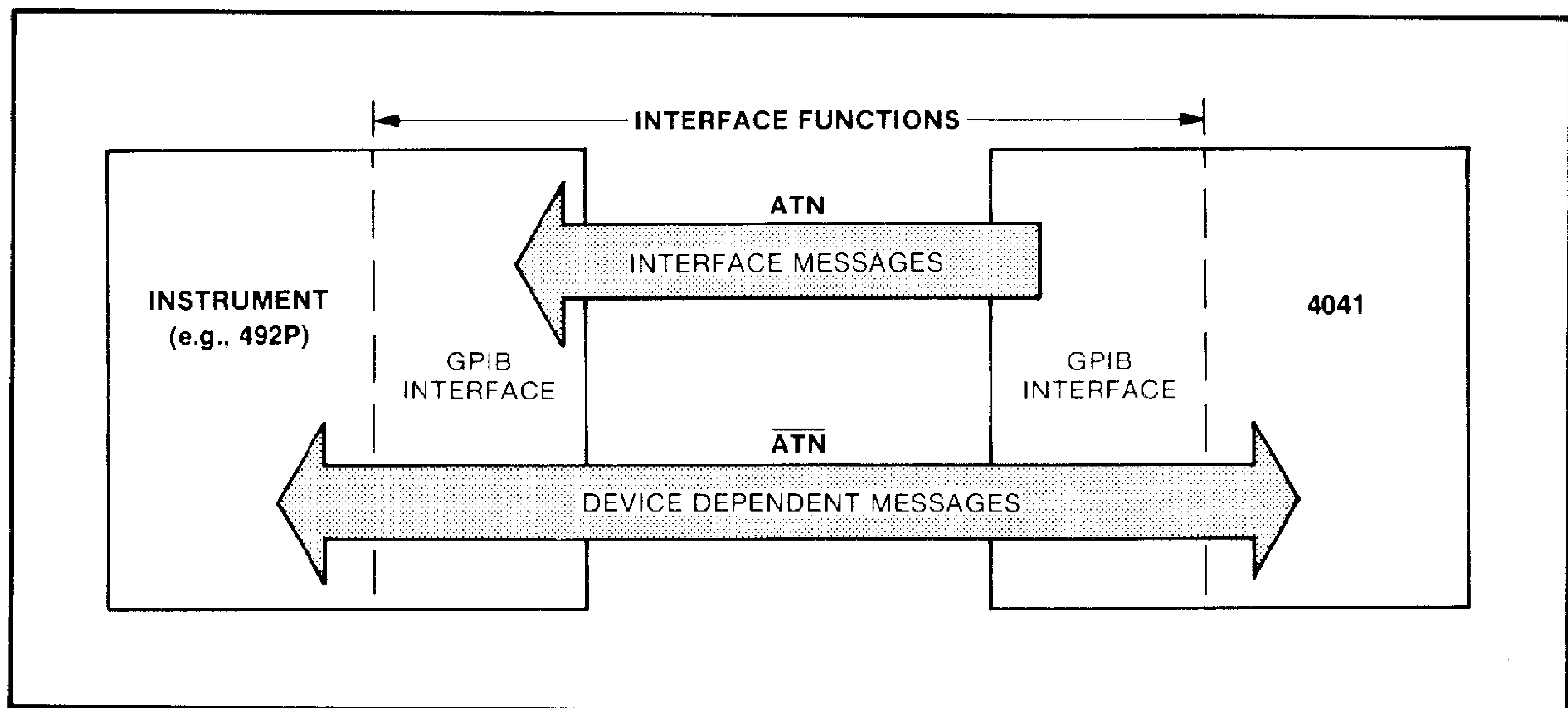


Fig. 1-3. *The controller sends interface messages with Attention (ATN) asserted. These messages control interface functions. Device dependent messages, sent with ATN unasserted, control instrument functions.*

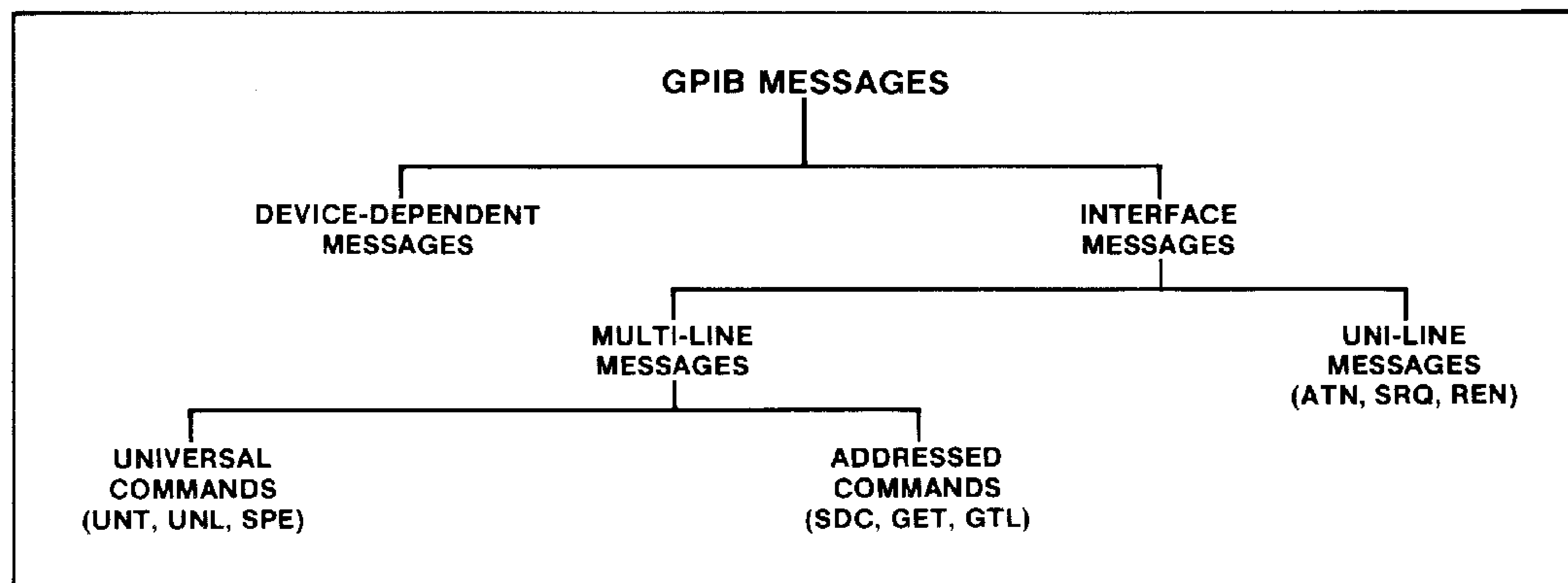


Fig. 1-4. *Messages sent over the GPIB can be divided into two general types—interface messages and device-dependent messages. Interface messages are further divided into universal multi-line messages, addressed multi-line messages, and uni-line messages.*

Transmitting and receiving data. Most instruments send data to 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 oscilloscope, such as the TEKTRONIX 7854, might receive waveform data from the controller for processing or display on its CRT. Data may be transmitted using a variety of codes including binary and 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 by polling the devices, reading a status byte from each instrument and taking the appropriate action.

Processing the Data

The third 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 DM 5010 Programmable Digital Multimeter and the 7854 Programmable Oscilloscope, can do some processing internally. But many can only send raw data to the controller, so the processing task is left entirely to the controller.

This processing may involve simple calculations on single values, such as voltage readings from a DMM. More advanced applications may require array processing such as signal averaging, or computing pulse parameters from an acquired waveform. Powerful high-speed microcomputers have made lengthy and complex calculations, once left to large mainframe computers, feasible even in a small GPIB system controller. 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 magnetic tape or disk, provides a convenient means of logging data or results. In addition, the controller can display measurement or processing results or it can print results on a printer or other hard copy device.

GPIB Capability— More than GPIB Compatibility

An efficient, powerful GPIB system requires more than just a computer with a 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

capable GPIB controller. There is a considerable difference between GPIB compatibility and GPIB capability!

The TEKTRONIX 4041 System Controller is a powerful, flexible, expandable IEEE 488 systems controller. The standard 4041 comes with 32K bytes of memory (between 19K and 25K are available to the user, depending on the configuration). Additional memory can be added up to a total of 512K bytes.

Though the basic unit is principally designed as an execute-only controller for use in systems where operators have minimal computer background, a variety of options and peripherals are available to equip the 4041 for full interactive program development and user flexibility.

The 4041's powerful hardware is supported by a highly enhanced BASIC language. BASIC is an excellent language for occasional programmers because it is simple and easy to learn. But, standard BASIC leaves much to be desired for many sophisticated progamming tasks. The enhancements provided by 4041 BASIC create an excellent environment for more advanced programmers. These enhancements include FORTRAN-like subprograms, parameter passing, local and global variables, a powerful debug mode, an extremely flexible I/O system, and several data types, create an excellent environment for more advanced programmers. The modularity afforded by true subprograms also allows a team of programmers to work on a single software task.

The 4041 was specifically designed as an instrument controller. Its powerful, flexible I/O structure makes handling virtually any GPIB device simple. In its default power-up condition, the 4041 implements the Tektronix Standard Codes and Formats so you can communicate instantly with Tektronix IEEE 488 instruments without worrying about command or numeric formats, delimiters, etc. In addition, a set of high-level commands are provided to implement most of the IEEE 488 functions, such as Device Clear (DCL) and Local Lock-Out (LLO).

The 4041 was designed as a GPIB controller—it has all the essential elements of a powerful, flexible and **capable** GPIB system controller (Fig. 1-5).

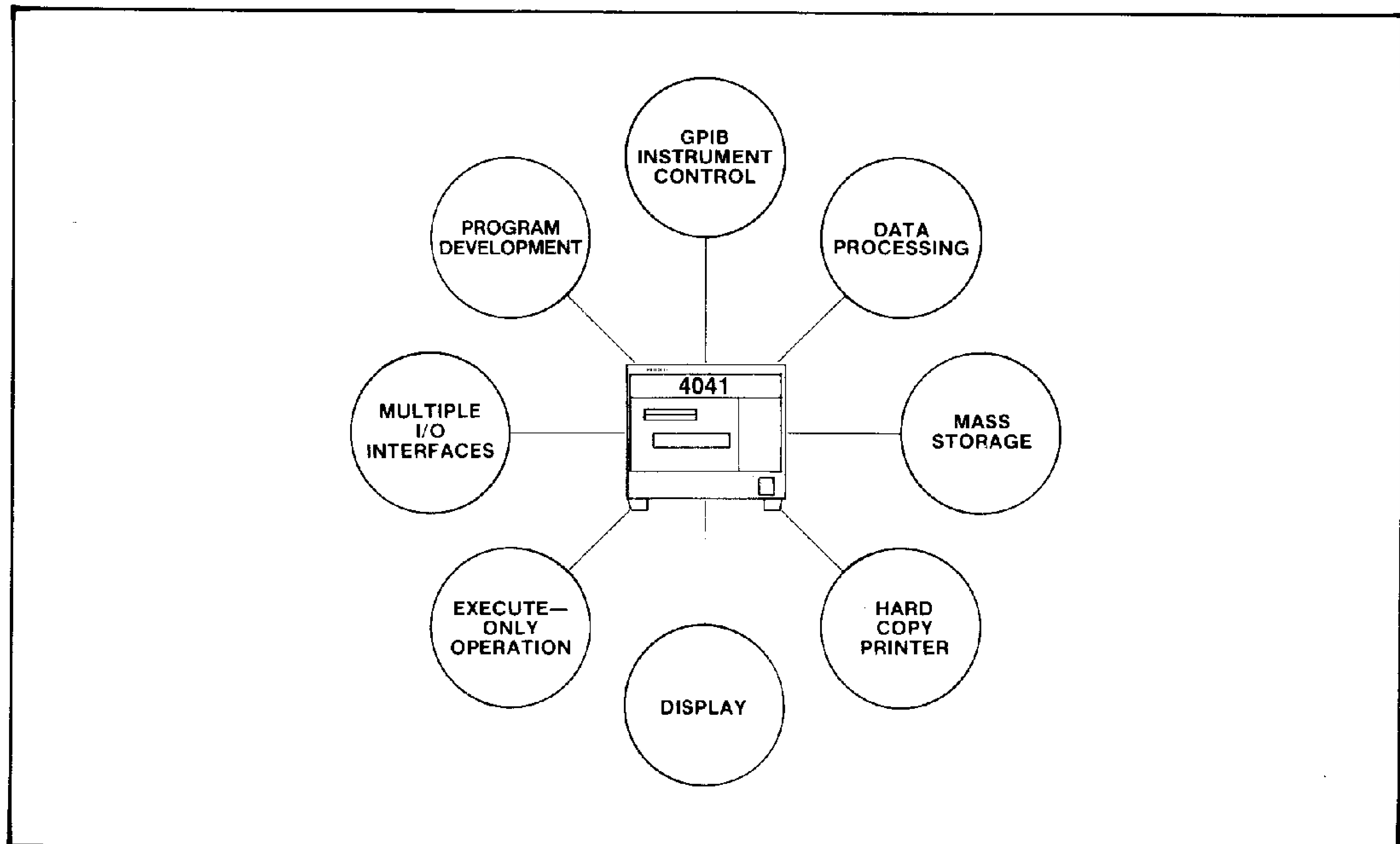


Fig. 1-5. *The 4041 is more than a GPIB-compatible computer—it's a capable GPIB system controller.*

Section 2 — Configuring a 4041 GPIB System

The GPIB is a flexible interface—it can efficiently link many different types of instruments together to perform a variety of jobs. Section 1 described 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:

- **What is the system's operating environment?** Will it be performing tests on a production line? If so, the operator will probably have minimal computer background. These operators usually need complete, detailed prompting and complete error handling. The software must be protected so that unexpected or incorrect inputs can't abort the program or "crash" the system. In this environment, a small, simple keypad with user-definable keys suits the need. In contrast, technicians and engineers often need a full-size keyboard with program editing functions.

- **Will the system 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 signal sources should be programmable.

- **Will the system acquire data?** If the system is intended to make automated measurements, some type of data acquisition is a given. This acquisition could be as simple as a DC voltage measurement, or as complex as a high-speed 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 4041 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.

- **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.

- **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 4041. Later, when the acquisition is complete, the 4041 can read and process data from the peripheral at a slower rate. It may even 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.

- **How will the system interface with the device under test?** Stimulus signals, power sources, and measurement signals may have to be switched or routed. In addition, parts handlers or other equipment may have to be interfaced to the system. Programmable multiplexers or multi-function interfaces may be required to provide interfacing to the device under test.

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. Remember, GPIB compatibility is not synonomous with GPIB programmability.

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."

- **How fast is the instrument?** Speed can be an important factor in choosing GPIB system components, particularly for systems intended for high-speed testing in a production environment. The speed of a GPIB instrument is determined by four basic factors: the time required for acquisition, internal processing, data transfer, and human interaction. 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 7 describes some techniques for estimating the performance of a GPIB system.

- **What interface functions are implemented?** A device's GPIB interface links the GPIB to the programmable device functions. The IEEE 488 standard allow designers 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—the ability to generate the handshake cycle for transmitting data.

ACCEPTOR HANDSHAKE—the ability to generate the handshake cycle for receiving data.

TALKER—the ability to transmit data on the bus.

LISTENER—the ability to receive data from the bus.

SERVICE REQUEST—the ability to request service from the controller via the SRQ line.

REMOTE/LOCAL—the ability to switch between local and remote operation.

PARALLEL POLL—the ability to report a single status bit during a parallel poll.

DEVICE CLEAR—the ability to be initialized by a bus command.

DEVICE TRIGGER—the ability to initiate an operation on receipt of a bus command.

CONTROLLER—the ability to act as the controller-in-charge.

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 triggerred by it (Fig. 2-1).

- **What's the address?** 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. 4041 BASIC adds this offset to the primary address automatically, so you only have to remember a single primary address. An instrument actually has one address for talking (if it

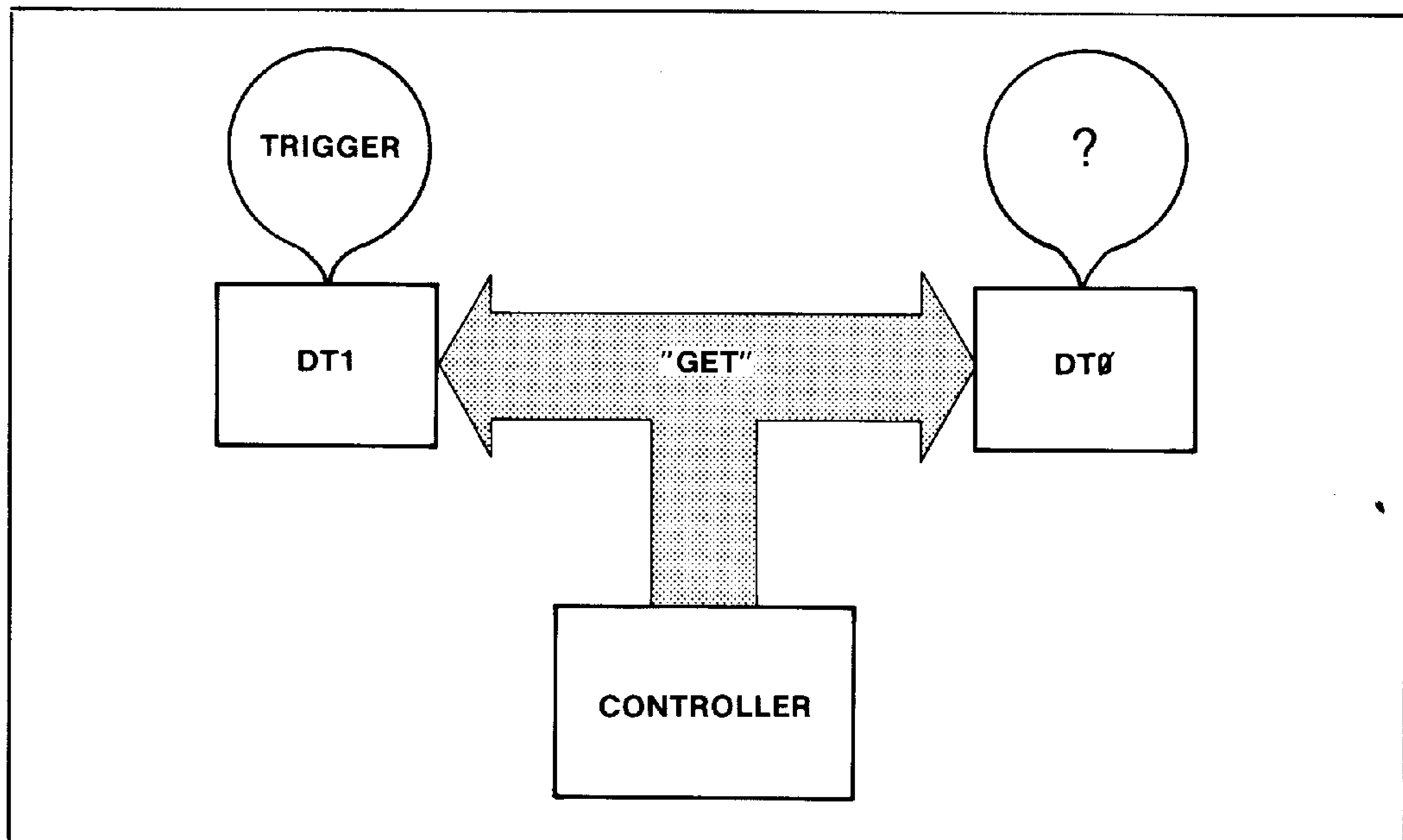


Fig. 2-1. *Some interface capabilities may not be useful if only one instrument in the system implements that capability. For example, the Device Trigger function may not be useful for synchronizing system operations if only one instrument implements the DT1 subset. Instruments with the DT0 subset ignore the GET message.*

can talk) and one address for listening (if it can listen). But the controller automatically generates these "absolute" talk and listen addresses from the single primary address.

In 4041 BASIC, the primary address is specified in the range of 0-30. 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-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 several 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.

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. 31 is not a valid address—it is used for the universal UNTalk and UNListen commands. Setting a device to address 31 effectively removes it from the bus since it can never be addressed. A typical set of address switches is shown in Fig. 2-2.

If a secondary address is required, it is usually set by a separate set of switches. 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 (See Table 2-1). 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. Some specific examples of how this is accomplished in 4041 BASIC are provided later.

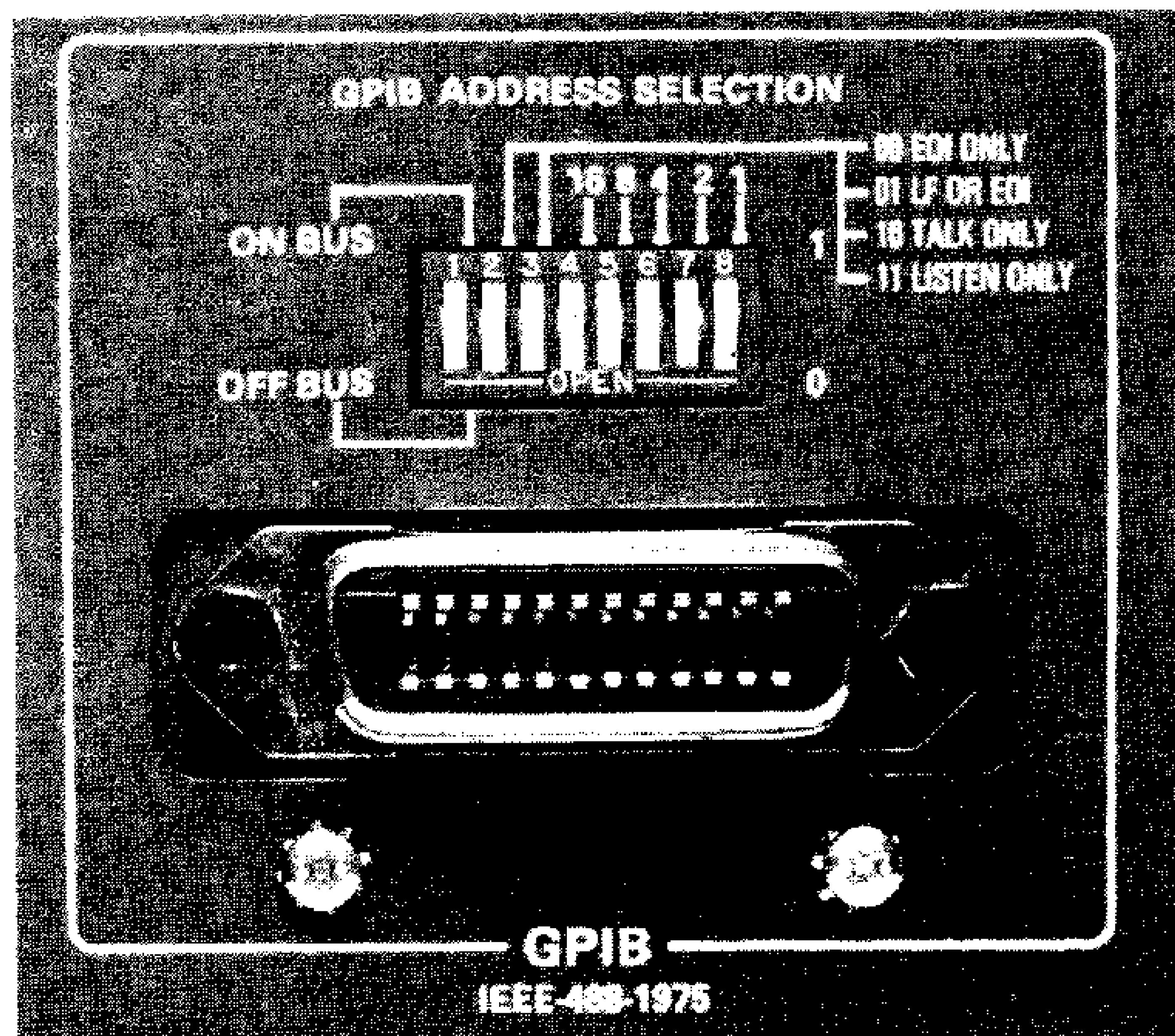


Fig. 2-2. *A typical set of GPIB address and message terminator selection switches. This set of switches also selects the ON BUS/OFF BUS conditions and the talk-only, listen-only, or talk/listen modes.*

TABLE 2-1
7612D AND 7912AD SECONDARY ADDRESSES

Plug-in Compartment	Secondary Address
Channel A plug-in	Mainframe secondary address + 1
Channel B plug-in	Mainframe secondary address + 2

- **Who's in charge here?** Two kinds of controllers are allowed on the GPIB: the system controller, and the controller-in-charge (often abbreviated as CIC). At any time, a GPIB system can have only one device acting as system controller and one device acting as controller-in-charge. The system controller and the controller-in-charge may be the same or different devices. The system may, however, have any number of devices that are capable of acting as controller-in-charge. Only one device may be the system controller and this task cannot be passed to other devices.

The system controller has some special privileges. It is the only device that can assert the

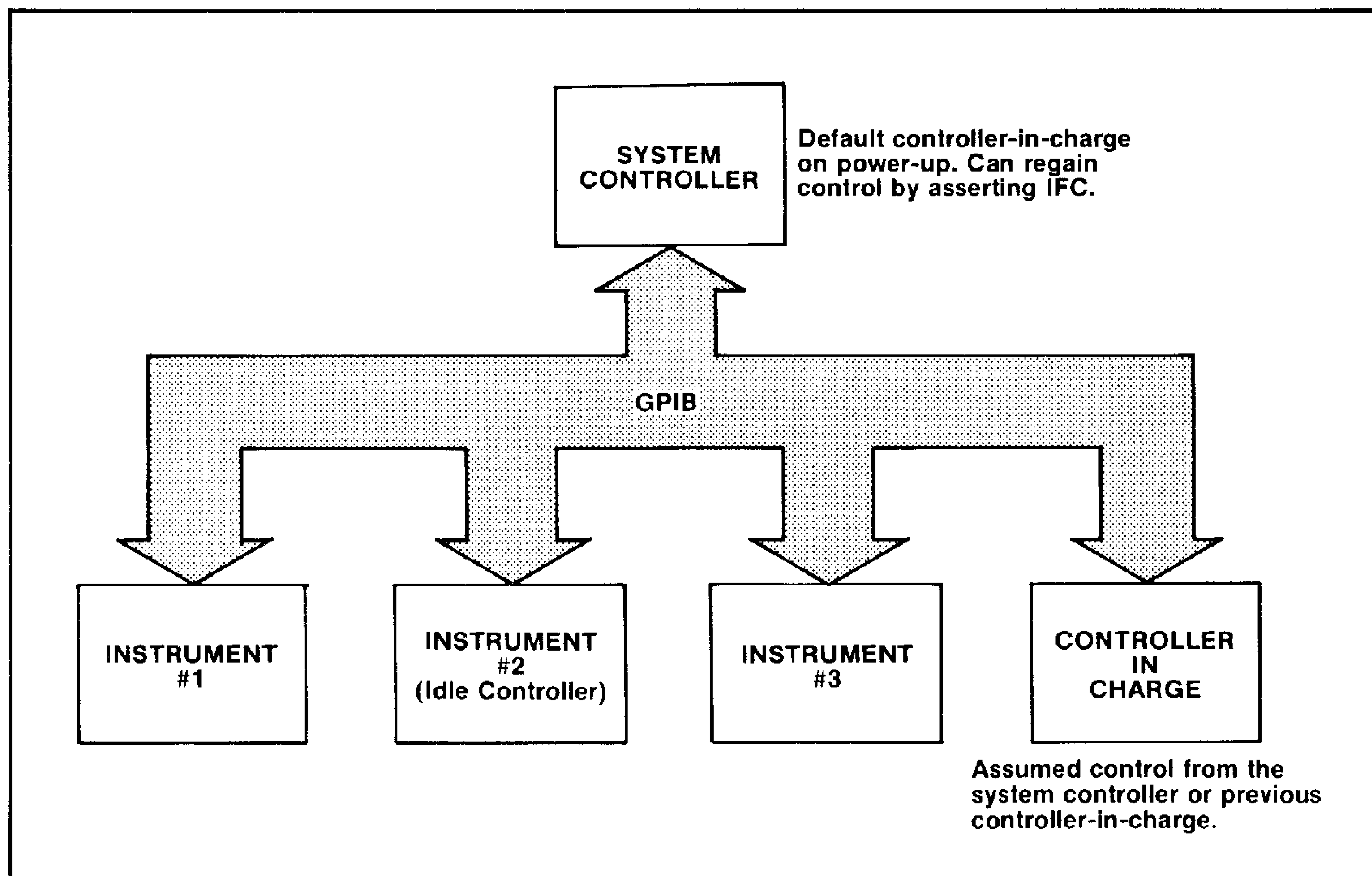


Fig. 2-3. *There can only be one system controller in the system and only one device can be the controller-in-charge at any time. However, there can be any number of devices capable of assuming control.*

IFC or REN (Remote ENable) lines. When the system is powered-up or the IFC line is asserted, the system controller becomes the controller-in-charge.

The system controller can pass control of the system to another device capable of assuming control of the bus. The new device becomes the controller-in-charge. The controller-in-charge can, in turn, pass control to another device capable of assuming control or it can pass control back to the system controller. The system controller can also regain control of the system by asserting the IFC line.

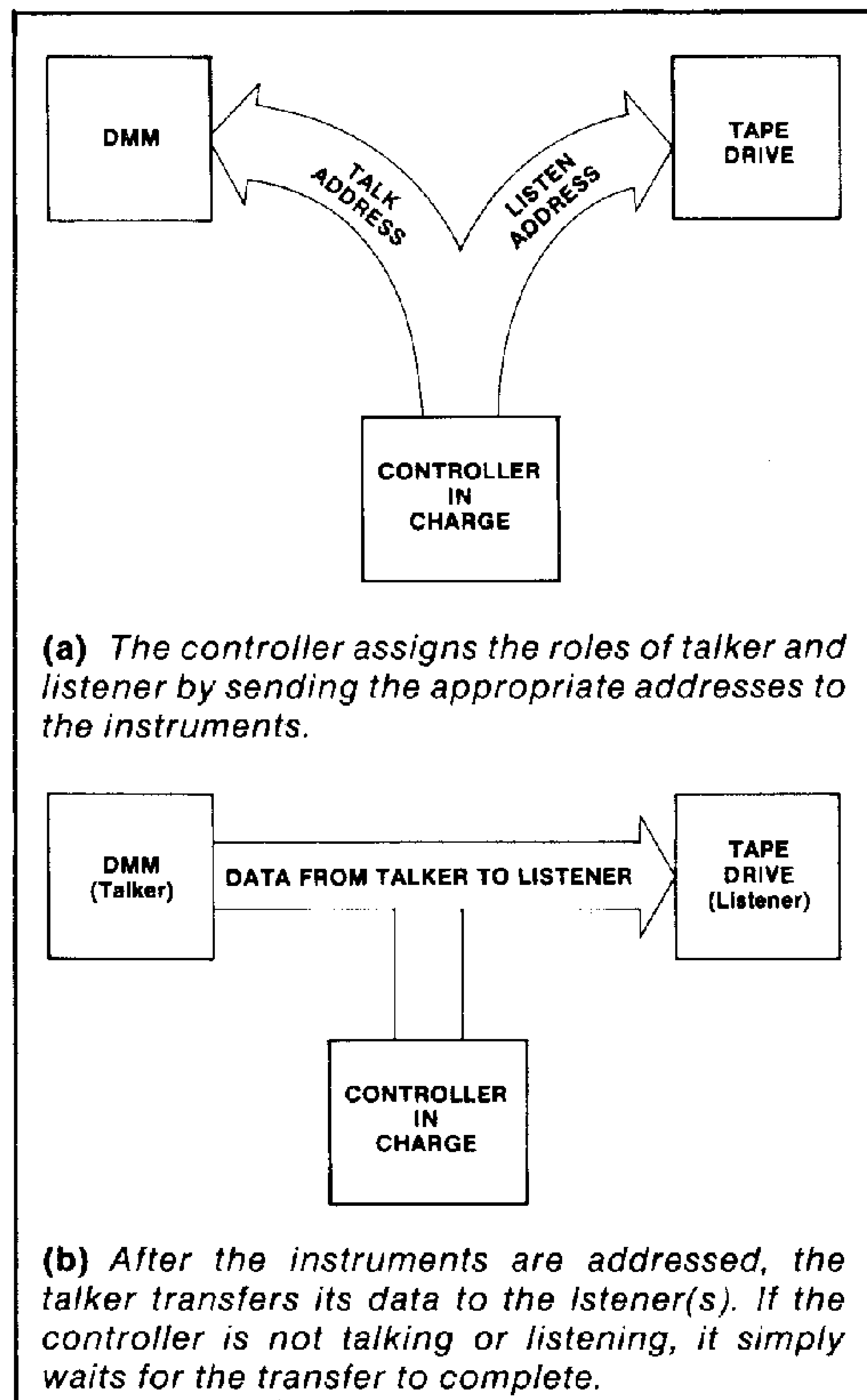
The controller-in-charge is the only device that is permitted to send messages with ATN asserted. These messages include addresses, and other interface messages. The controller-in-charge is responsible for controlling all bus transfers and handling interrupts. Figure 2-3 illustrates the relationship between the system controller and the controller-in-charge.

Some devices capable of acting as system controller do not have the capability of passing control to another controller. It is important to note this distinction because devices that cannot pass control will not operate on the bus unless they are the system controller. If you intend to operate the system with more than one controller it is important to check for the capability to pass control.

• **Who's talking, who's listening, and who's controlling?** Devices in the system can take three roles: Talker, Listener, or Controller. The controller-in-charge assigns the roles of the other devices in a system. Once the assignment of roles (addressing) is complete, the device assigned as the talker sends data to the devices assigned as listeners. For example, the controller might tell a DMM to talk and a tape drive to listen. When the data transfer begins, the DMM sends data and the tape drive receives it (Fig. 2-4).

There can only be one controller-in-charge and one talker at a time, though there can be several listeners. The 4041 can take on any of these roles. There can even be several 4041's in a system—one performing as a controller and another acting as a smart data logger, for example.

A single 4041 can also take different roles on each of it's two GPIB ports (with its optional second GPIB port). On one bus, the 4041 can act as the system controller or controller-in-charge, while it acts as a talker/listener on the other bus. The two GPIB interfaces in the 4041 with Option 1 are entirely independent.



(a) *The controller assigns the roles of talker and listener by sending the appropriate addresses to the instruments.*

(b) *After the instruments are addressed, the talker transfers its data to the lstener(s). If the controller is not talking or listening, it simply waits for the transfer to complete.*

Fig. 2-4. *The controller-in-charge assigns the roles of talker and listener to devices in the system. The controller may or may not be involved in the data transfer.*

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. 2-5). 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. In addition, if the system will have more than one controller, be sure that they can "pass control." In other words, be sure the controller can let another device take over the role of controller-in-charge.

• **What is the message format?** Another important consideration when you are configuring a GPIB system is the message format used by each instrument. The IEEE 488 standard specifies the mechanical, electrical and functional aspects of the interface, but it does not specify the content or syntax of the messages transferred across this interface. As a result, devices connected by the IEEE 488 bus have a compatible hardware interface, but there is no guarantee that they will speak the same language.

The telephone system provides a good analogy of this problem. Telephones provide a compatible hardware interface that makes it possible to call just about any telephone in the world. But if the people on each end don't speak the same language, no meaningful communication can take place even though they can hear each other. In the same way, two devices connected by the IEEE 488 bus may not be able to communicate if they don't speak the same language even though they have a compatible hardware interface. Figure 2-6 illustrates this concept.

Since IEEE 488 doesn't specify the syntax or coding of the messages, there are a variety of codes and syntax formats used by various manufacturers. This can be a source of frustration when programming a system, because the programmer may have to remember several different message formats.

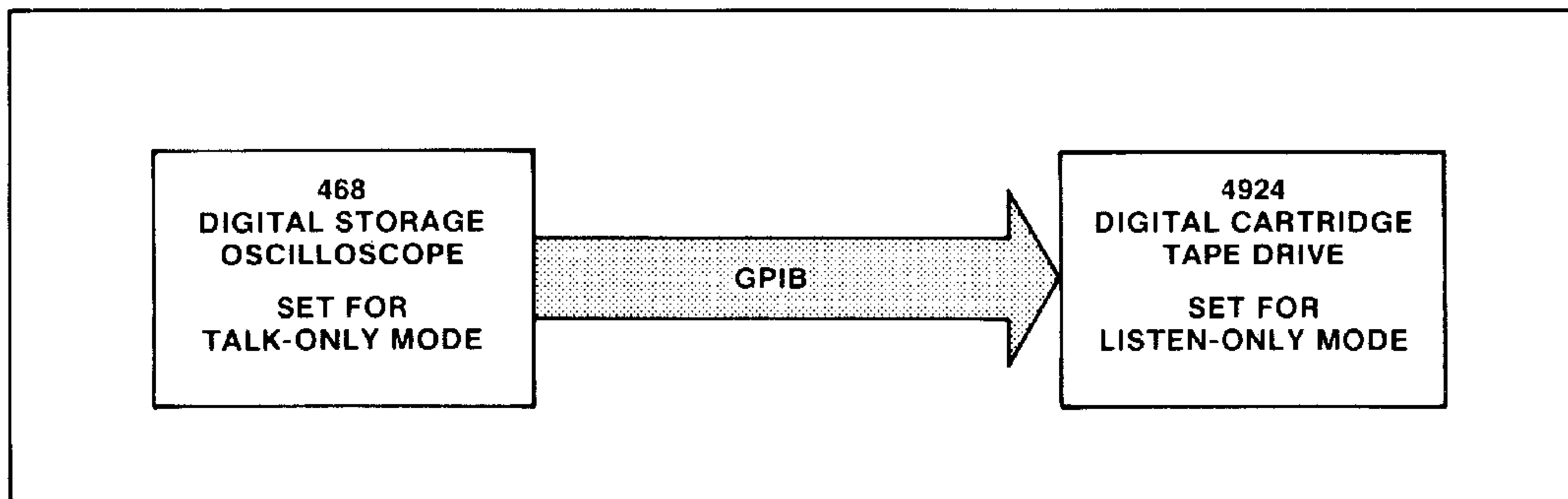


Fig. 2-5. *Some instruments can be manually set to permanent talker (talk-only mode) or permanent listener (listen-only mode). This allows small systems, such as the 468/4924 system shown here, to operate without a controller. These instruments may also be operated with a system controller.*

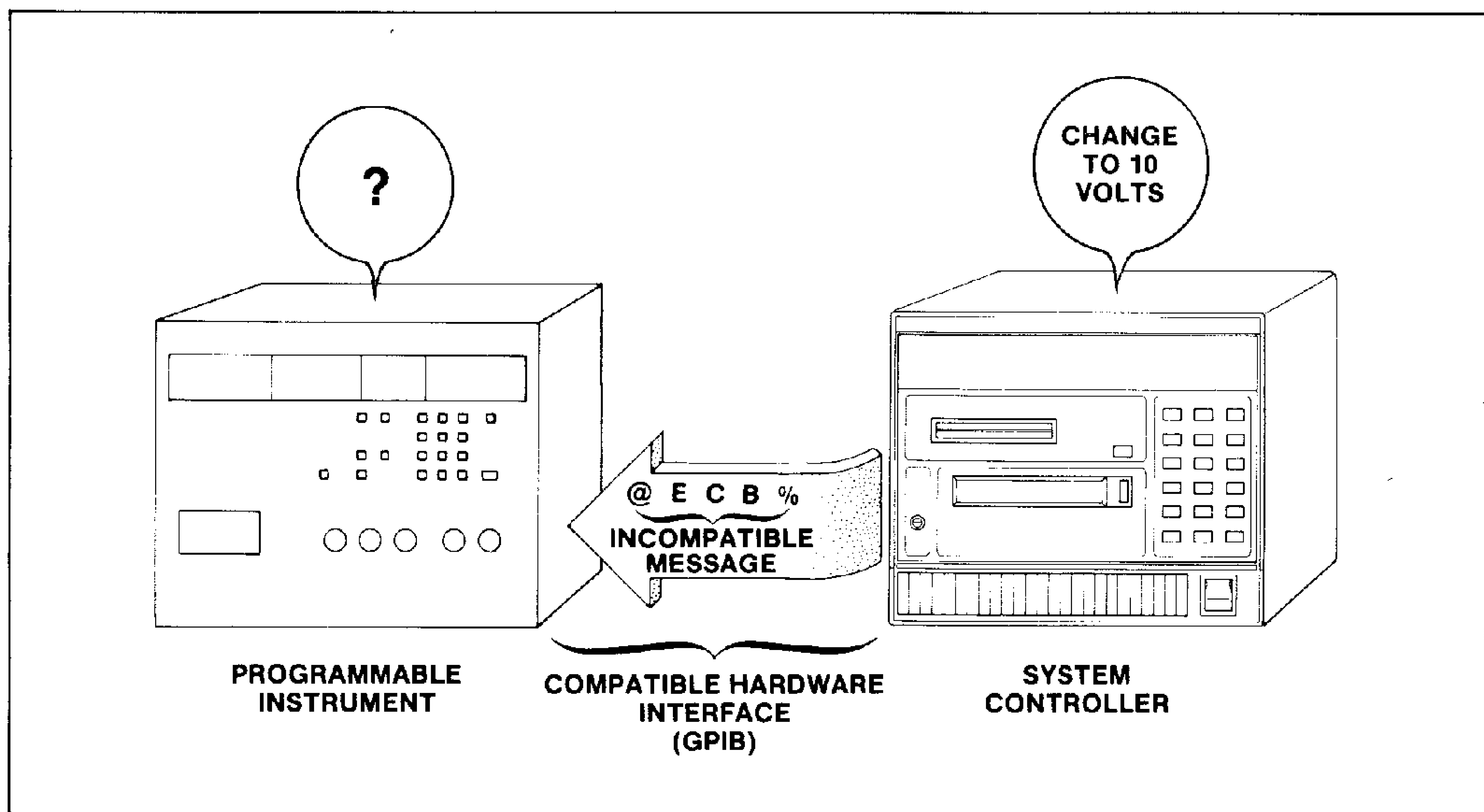


Fig. 2-6. *The controller and instruments need more than a compatible hardware interface; they need a compatible message structure to communicate.*

Tektronix has developed a standard for codes and formats 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 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.

A brief summary of the Tektronix Standard Codes and Formats is provided in Appendix B.

- **What is the message terminator?** 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 standard codes and formats, which specifies EOI as the message terminator. Most Tektronix instruments can be also be set to use the line-feed terminator when operated with equipment from other manufacturers. In addition, Tektronix controllers can be set to use any character in addition to EOI as the message terminator.

Getting It Together

Once you understand the capabilities and requirements for each instrument in your system, the job of actually configuring the system should be simple. This section provides a few guidelines for connecting the instruments together and setting bus addresses.

Setting the bus address. The first step is setting the primary bus address and, if used, secondary addresses for each instrument. Remember that every device must have a unique address. Valid primary addresses are 0-31, but selecting address 31 logically removes the device from the bus; it does not respond to any addresses and it remains both unlistened and untalked. In addition, no instrument on the bus can be set to the same address as the 4041. The 4041 powers-up with a default address of 30, but this value can be changed under program control using the SET DRIVER command.

It is not necessary to arrange the addresses in any particular order. As you set the addresses, write them down for reference when writing programs. Most Tektronix instruments can display their current address setting by pressing a front-panel button.

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

With option 01, the 4041 has two GPIB ports. Each port can support an independent system. Addresses on each bus must be unique, but there can be duplicate addresses on different busses. The 4041 occupies address 30 on both busses unless its address is changed in the program. Notice that the 4041 can occupy different addresses on the standard GPIB bus (GPIB0) and the Option 1 bus (GPIB1). The 4041's address is set with the MA (My Address) parameter in a SET DRIVER statement.

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. The 4041 uses EOI as the message terminator, but any other character (including line feed) can also be defined as an alternate delimiter using the EOM (End Of Message) parameter in an OPEN statement. If an alternate delimiter is defined, the 4041 sends the EOM character at the end of the message and asserts EOI with that character. On input, the EOM character **or** EOI are accepted as terminators.

Cabling the instruments. The next step is cabling the instruments together. Up to 15 devices (including the controller), 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. Some devices, like the Tektronix TM 5000-series of modular instruments are housed in a mainframe, but each device counts as a bus load (Fig. 2-7). Check the instrument manuals for more details.

With option 01, the 4041 has two GPIB ports, so up to 14 devices can be connected to each bus. (The 4041 counts as the 15th device on each bus.) Since

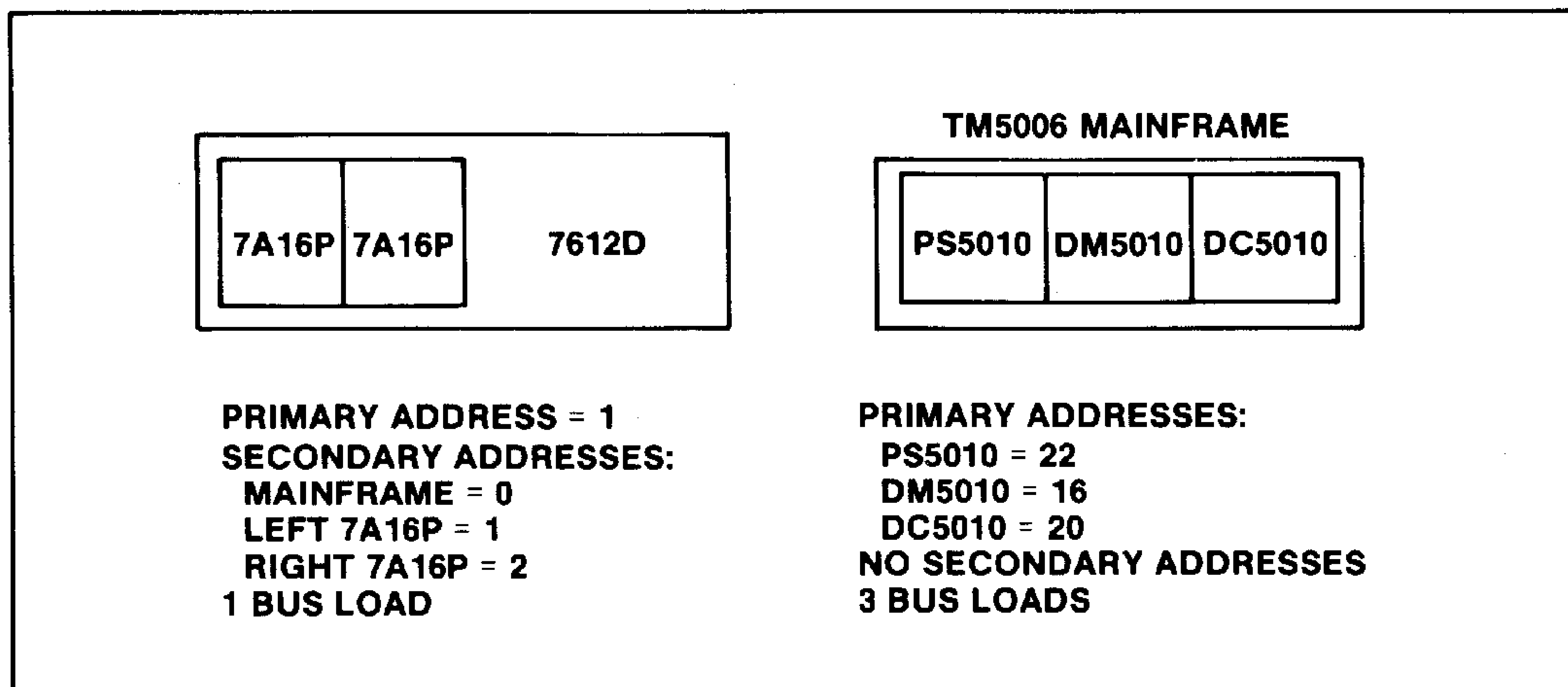


Fig. 2-7. *Some instruments, like the 7612D Programmable Digitizer, use a single primary address and represent one bus load. Secondary addresses select the plug-ins or mainframe. Other devices, like the TM 5000-series of programmable instruments represent one bus load per instrument. Each instrument has a separate primary bus address.*

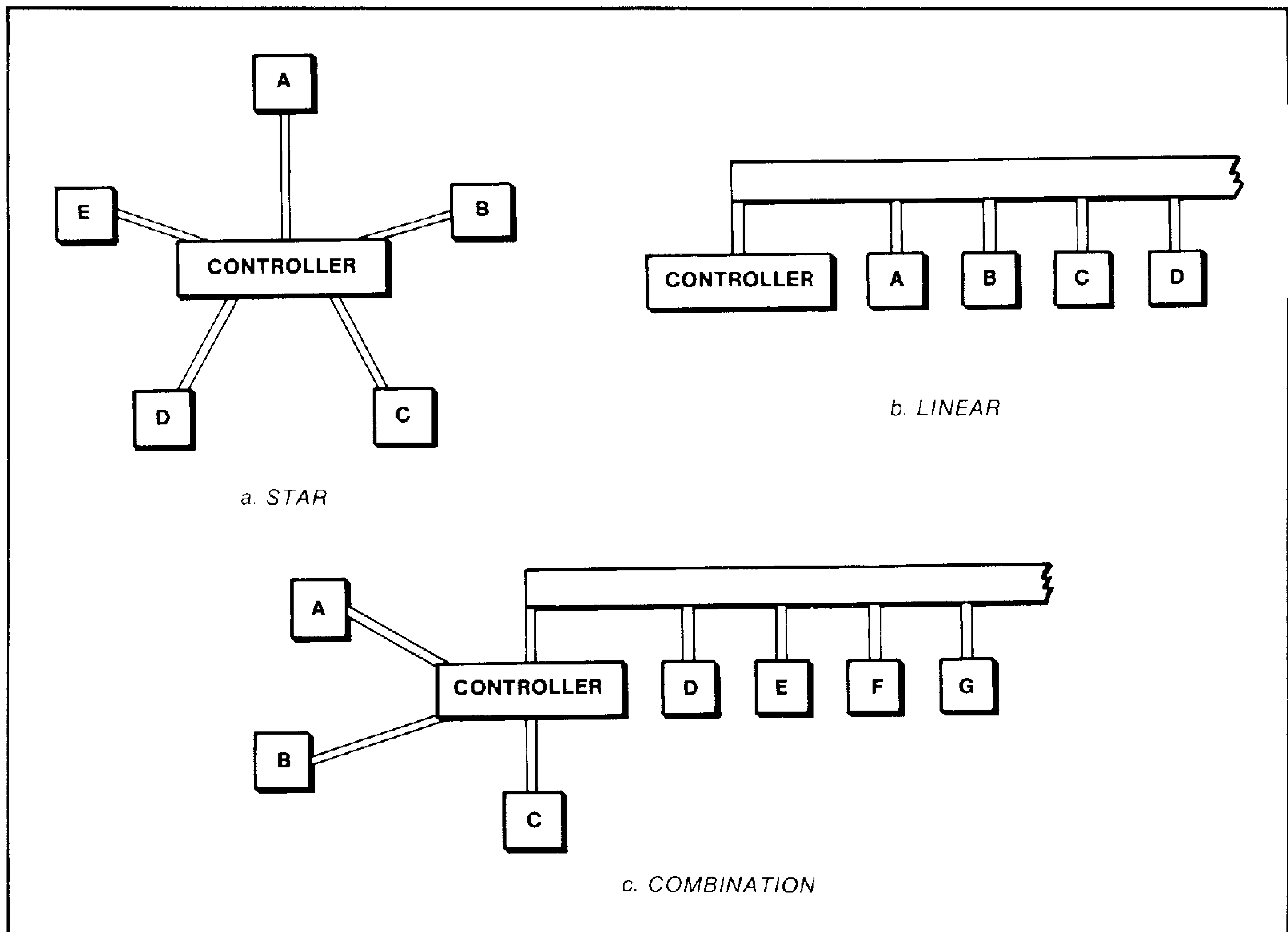


Fig. 2-8. *The GPIB system can be cabled in a star or linear configuration or a combination of the two.*

the option 1 GPIB port on the 4041 supports DMA (Direct Memory Access), the system data transfer speed may be improved by arranging instruments on the two busses so that the faster instruments are on the option 1 port (GPIB 1) and the slower ones are on the standard port (GPIB 0). The DMA data transfer mode is especially advantageous with instruments that transfer large amounts of data, such as waveform digitizers.

The system can be cabled in a star or linear configuration (Fig. 2-8). 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 further 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 IEEE 488 standard for more details.

In Summary

Figure 2-9 shows a checklist summarizing the considerations for selecting system components and configuring the system.

- ☐ What is the system's operating environment?
- ☐ What is the operator's skill level?
- ☐ Will the system generate test stimuli?
- ☐ Will the data need to be processed?
- ☐ Will data or test results be logged to a peripheral device?
- ☐ Is the instrument really programmable?
- ☐ How fast is the instrument?
- ☐ What interface functions are implemented?
- ☐ What's the address?
- ☐ Who's talking, who's listening, and who's controlling?
- ☐ What is the message format?
- ☐ What is the message terminator?

Fig. 2-9. *Component selection and system configuration checklist.*

The 4041 System Controller provides the hardware features required in a powerful, flexible, and configurable GPIB controller. But the key to the 4041's power and flexibility is in it's BASIC operating system software. This section introduces you to 4041 BASIC. It is not intended to be a tutorial on 4041 BASIC. Instead, it provides a brief overview of the extensions and special features of 4041 BASIC. A more detailed description of the 4041 BASIC I/O system is also provided.

4041 BASIC Enhancements

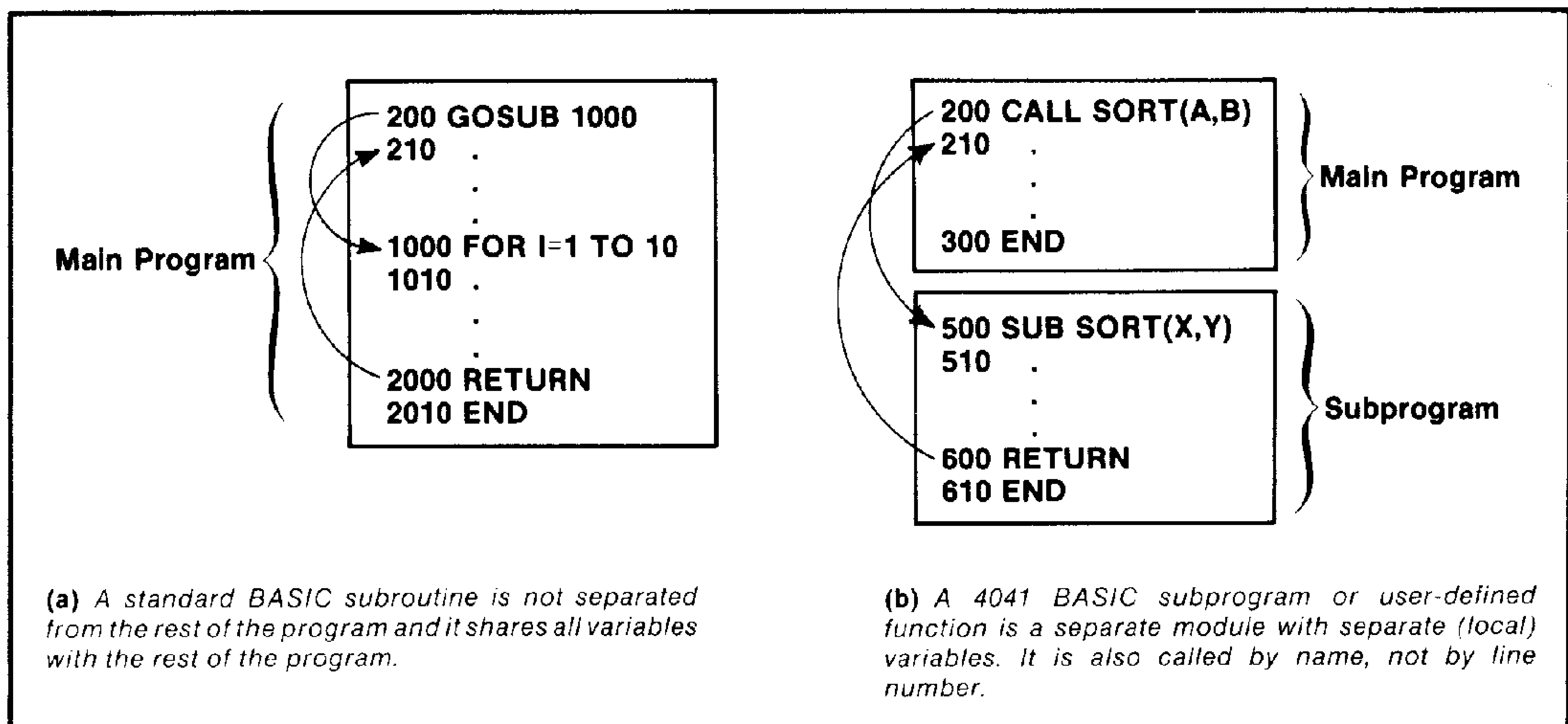
4041 BASIC combines the simplicity and ease of use of standard BASIC with enhancements that overcome many of the limitations of standard BASIC. The major enhancements are described below.

• **Subprograms and user-defined functions.** In addition to the GOSUB statement provided in standard BASIC, 4041 BASIC provides FUNCTION, SUB, and CALL statements that allow you to set up full FORTRAN-like subprograms and user-defined functions. The standard BASIC subroutine is simply a set of lines that are called with a GOSUB and are terminated with a RETURN statement (Fig. 3-1). The subroutine is called by its starting line number and it is not separated from the rest of the program. All variables used in the subroutine are shared with the main program and there is no facility for passing arguments to the subroutine except through shared variables.

The 4041 SUB statement allows you to set up a subprogram that is like a subroutine with some special advantages. The subprogram is separated from the main program so that it can have independent variables. In addition, the subprogram can receive arguments and return results through the CALL statement. The subprogram is called by name, not by line number.

User defined functions are like subprograms in that they are independent routines called by name. However, they return a single value through the function name itself. The FUNCTION statement declares the beginning of a FUNCTION routine. The function is called just like a reference to a standard BASIC function.

The subprogram and user-defined functions make modular program design much simpler and allow a team of programmers to work on individual parts of a larger task with a minimum of interaction.



(a) *A standard BASIC subroutine is not separated from the rest of the program and it shares all variables with the rest of the program.*

(b) *A 4041 BASIC subprogram or user-defined function is a separate module with separate (local) variables. It is also called by name, not by line number.*

Fig. 3-1. *4041 BASIC provides a sophisticated FORTRAN-like subprogram and user-defined function capability that makes modular programming simple.*

- **Local and global variables.** Variables in a 4041 subprogram or user-defined function can be declared as "local." Local variables are defined only within the subprogram or user-defined function, so variables names can be duplicated in the subprogram and the main program. Local variables in a subprogram or user-defined function are completely independent of other variables outside the subprogram even if variables in the other program segments have the same name.

- **Descriptive variables names.** Variable names in 4041 BASIC can be up to eight characters long. As a result, programs are easier to read and understand because variable names can more clearly represent their function. For example, a variable that stores the maximum value of an array can be called ARRAYMAX instead of M or M1.

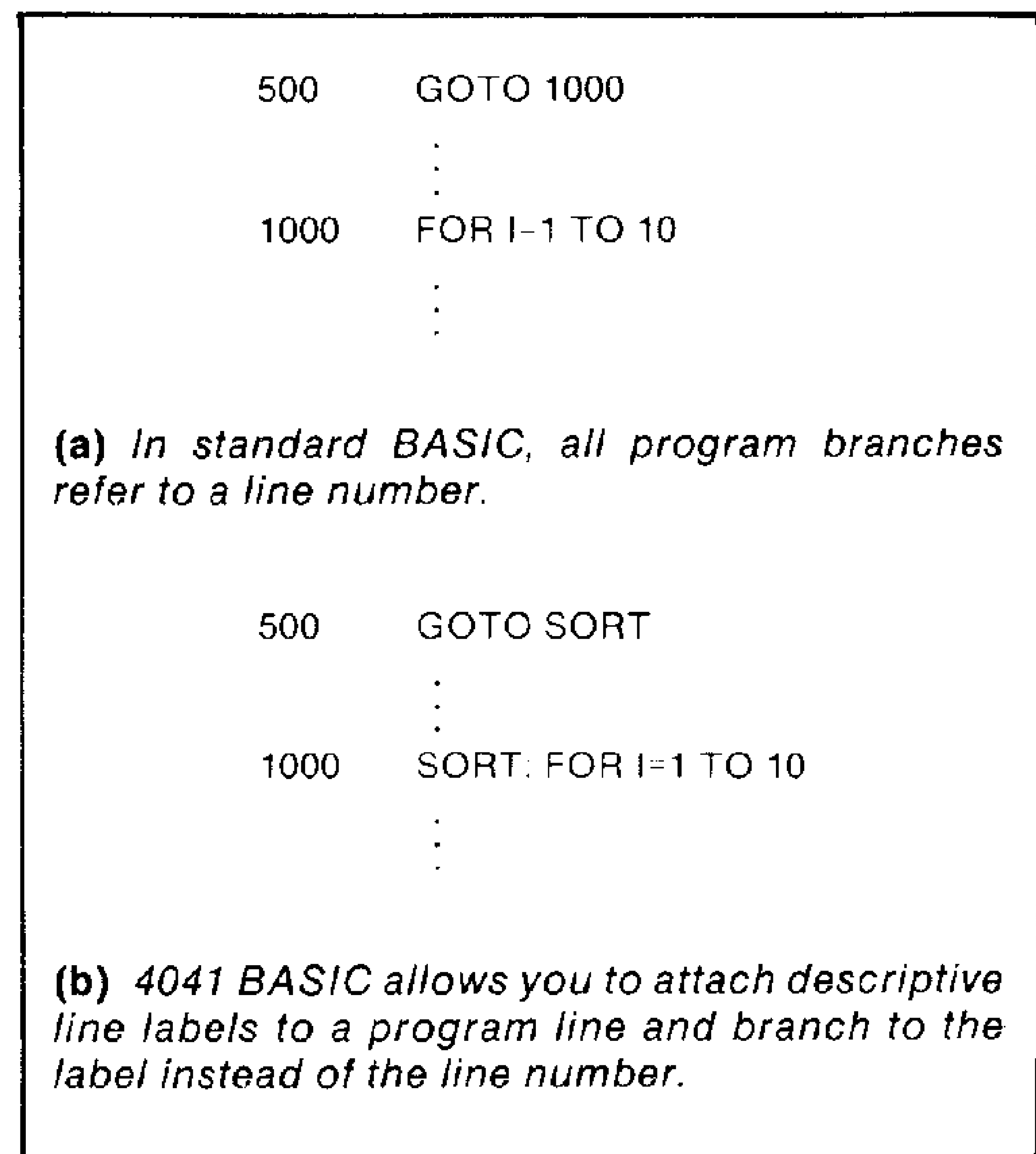
- **Variety of data types.** A variety of data types are available in 4041 BASIC to match the application. Numeric data can be stored in one of three formats: integer, short floating point, and long floating point. Numeric data and strings can also be stored in arrays.

The choice of which data format to use depends on accuracy requirements, memory consumption, and execution speed. The variety of data formats available in 4041 BASIC allows you to match the data format to your application.

- **Line labels.** Program lines in a 4041 BASIC program can be labeled and lines can be referenced by label, instead of by line number. Programmers don't need to remember line numbers and line labels can describe the code that follows. SORT, for example, might refer to a sort routine. The program jumps to the SORT routine with a GOTO SORT statement instead of GOTO 1000 (Fig. 3-2).

- **Debugging aids.** To simplify the task of debugging programs, 4041 BASIC includes a debug mode that allows the programmer to trace each change of a specified variable during program execution, trace program flow, and set "breakpoints" to temporarily halt program execution.

- **Powerful GPIB control.** Since the 4041 was primarily designed as an instrument controller, several enhancements are included in 4041 BASIC for efficient GPIB system control. Commands are provided for simple ASCII and binary communication. In addition, most of the IEEE 488 interface messages are implemented as simple high-level keywords.



(a) *In standard BASIC, all program branches refer to a line number.*

(b) *4041 BASIC allows you to attach descriptive line labels to a program line and branch to the label instead of the line number.*

Fig. 3-2. *4041 BASIC allows line labels that make programs easier to read and maintain.*

- **Tek Codes and Formats support.** The 4041 is designed to be especially easy to operate with instruments that support the Tektronix Standard Codes and Formats. The Codes and Formats standard specifies simple consistent commands and syntax for all Tektronix GPIB instruments.

- **Flexible I/O structure.** The I/O structure of 4041 BASIC makes input and output operations simple. You can define the characteristics of the device once and assign a logical unit number to that device. Then I/O to the device is performed through the logical unit number. The logical unit number can be specified as either a constant or as a variable.

- **Proceed mode I/O.** 4041 BASIC allows you to start an I/O operation and proceed with other tasks while the I/O is in progress. An interrupt signals the program when the I/O is complete. Proceed mode can increase system performance by overlapping the I/O and other tasks—reducing the total time required for a measurement.

Input/Output in 4041 BASIC

I/O statements. A variety of I/O statements are provided in 4041 BASIC. These statements generally fall into four catagories: I/O control, high-level data transfer, low-level data transfer, and special-purpose I/O statements. I/O control statements set I/O parameters or control how I/O is performed. High-level I/O statements transfer formatted data such as ASCII characters or numbers or binary values. Low-level statements transfer 8-bit values without formatting. Special-purpose I/O statements perform functions required by a particular device, such as formatting a DC-100 tape, or polling a GPIB device. Table 3-1 lists the statements in each class.

I/O devices. All input and output in the 4041 is directed to one of five devices (seven with option 1). Each of these devices is represented in 4041 BASIC by a mnemonic that identifies the device. The devices and their mnemonics are listed in Table 3-2.

TABLE 3-1
I/O STATEMENTS

I/O Control Statements
OPEN - associates a logical unit with a stream spec.
CLOSE - returns a logical unit to a default stream spec.
SELECT - specifies default stream spec for RBYTE, WBYTE and POLL.
RESTORE - resets DATA statement pointer to first element.
IMAGE - specifies data format for PRINT USING and INPUT USING.
SET CONSOLE - sets the console device to front-panel or the COMM port.
SET DRIVER - sets physical characteristics of a port.
SET PROCEED - sets or clears proceed mode for I/O operations.
High-level I/O Statements
COPY - transfers data from one device or file to another.
INPUT - transfers data from a device into memory.
PRINT - transfers data from memory to a device.
Low-level I/O Statements
RBYTE - transfers unformatted 8-bit bytes from a device into memory.
WBYTE - transfers unformatted 8-bit bytes or GPIB control commands from memory to a device.
Special-Purpose I/O Statements
GETMEM - transfers data from a buffer string into variables.
PUTMEM - transfers data from variables into a buffer string.
POLL - performs a serial poll on the GPIB.
APPEND - gets a program from a device and adds it to the program in memory.
LOAD - clears the current program and loads a program from a device.
SAVE - sends a program stored in memory to a device.
LIST - sends a listing of a program stored in memory to a device.
FORMAT - formats a specified device to prepare it to receive data.*
DELETE FILE - deletes a file on the specified device.*
RENAME - renames a file on the specified device.*
DIR - prints a directory of the specified device on the console or another device.*

** The default device for these commands is the SYSDEV (System Device) which defaults on power-up to the tape. The System Device can be changed using the SET SYSDEV command.*

TABLE 3-2
4041 I/O DEVICES AND MNEMONICS

Mnemonic	Device
FRTP	Front panel (LED display, front-panel keypad, and program development keyboard)
COMM or COMM0	Standard RS-232C port
COMM1	Option 1 RS-232C port
GPIB or GPIB0	Standard GPIB port
GPIB1	Option 1 GPIB port
TAPE	Internal DC-100 tape drive
PRIN	Internal 20-character thermal printer

I/O statements, such as PRINT, default to one of these devices if you don't explicitly specify a device in the statement. For example, the PRINT statement transfers data to the console device (the front-panel LEDs or the COMM port) if a device is not specified. Table 3-3 shows the I/O statements and their default devices.

TABLE 3-3
DEFAULT DEVICES FOR I/O STATEMENTS

Statement	Default Device
PRINT	Console (FRTP or COMM)*
INPUT	Console (FRTP or COMM)*
RBYTE	GPIB0
WBYTE	GPIB0

* *Console device is set with a SET CONSOLE statement; default is FRTP.*

The system console device. The primary device through which a user communicates with the 4041 is called the system console device or "console." The default PRINT and INPUT statements transfer data to and from the console device. In addition, on a 4041 equipped with the program development ROMs (Option 30), programs are input and edited from the console device.

At power up, the console device defaults to the front panel (FRTP) which includes the front panel LEDs and keypad and the program development keyboard (with Option 31). The console device can be changed to an RS-232C device connected to one of the COMM ports with a SET CONSOLE statement such as:

```
Set console "COMM:"
```

NOTE

In order to use an RS-232C terminal as the system console device with a 4041 NOT equipped with Options 30 and 31 (Program development ROMS and keyboard), the user must have a DC-100 tape containing a file named "AUTOLD" that includes the statement SET CONSOLE "COMM:". The tape must be inserted at power-up or the tape can be inserted and executed with the AUTOLOAD button.

The system device. The 4041 has another "psuedo-device" called the system device (SYSDEV). On power-up the system device is the tape drive. All DIR, RENAME, and other file control statements default to the system device. The system device can be changed with the SET SYSDEV command.

Stream specifications. 4041 BASIC provides a means of specifying an alternate path for I/O other than the default device shown in Table 3-3. For example, output from a PRINT statement can be directed to a GPIB device instead of the front-panel. This alternate path is specified in a "stream specification." The stream specification (or stream spec, for short) defines the alternate device for the I/O statement as well as the characteristics of the device or transfer. A GPIB stream spec, for instance, can specify the bus address of the device and the end-of-message terminator character to be used in the data transfer.

The stream spec is a string constant or string variable containing the device mnemonic followed by the characteristics of the device or transfer. Consider, for example, a GPIB device connected to the standard GPIB port (GPIB0). The device is set for primary address 4 and secondary address 0 and the end-of-message (EOM) character is line feed (ASCII decimal code=10). The stream spec for this device would be:

```
"GPIB0(pri=4,sec=0,eom=<10>):"
```

If you want to send a voltage value stored in the variable VOLTS, you simply add the above stream spec to a standard PRINT statement as shown below.

```
Print #"GPIB0(pri=4,sec=0,eom=<10>):":volts
```

The stream spec can also be assigned to a string variable and the variable can be referenced in the PRINT statement like this:

```
100    Stream1$="GPIB0(pri=4,sec=0,eom=<10>):"
110    Print #stream1$:volts
```

Logical unit numbers. Instead of requiring you to enter the entire stream spec for every I/O statement, 4041 BASIC provides a shorthand way of referring to a stream spec. The stream spec can be assigned to a "logical unit number" with an OPEN statement as shown below.

```
Open #4:"GPIB0(pri=4,sec=0,eom=<10>):"
```

This statement associates the stream spec with logical unit number 4. From this point on, referring to logical unit 4 automatically uses the stream spec defined in the OPEN statement. As a result, the PRINT statement reduces to:

```
Print #4:volts
```

The logical unit number does not have to correspond to the primary address. Logical unit numbers can range from 0 to 32,767.

```
PRINT #"GPIB(PRI=4,SEC=0,EOM=<10>):":VOLTS
```

(a) *The stream spec can be inserted in a standard PRINT statement as a string constant.*

```
100    STREAM1$="GPIB(PRI=4,SEC=0,EOM=<10>):"
110    PRINT #STREAM1$:VOLTS
```

(b) *The stream spec can be stored in a string variable and the variable referenced in a standard PRINT statement.*

```
100    OPEN #4:"GPIB(PRI=4,SEC=0,EOM=<10>):"
110    PRINT #4:VOLTS
```

(c) *The stream spec can be associated with a logical unit number in an OPEN statement. Then the logical unit number is referenced in the PRINT statement*

Fig. 3-3. *There are three ways to use a stream spec to define an alternate I/O path.*

Figure 3-3 shows the four ways to use a stream spec—as a string constant in the PRINT statement, as a string variable, with a logical unit number specified as a constant, or with a logical unit number speicified as a variable.

I/O parameters. 4041 BASIC provides several "settings" that control the way I/O operations are performed. These parameters can be divided into three types:

- **Logical parameters**—set the characteristics of a particular device or data transfer. The bus address of a GPIB device and the end-of-message character are logical parameters.

- **Physical parameters**—set the characteristics of all I/O on a GPIB or COMM port. The COMM port baud rate is a physical parameter.

- **ASK$ parameters**—cannot be set, but return values that indicate the status of a device. The ASK$("LU",3) function returns the stream spec associated with logical unit number 3.

Logical parameters are set as part of a stream spec included in an OPEN or I/O statement. For example, to set up a transfer to a GPIB device at primary address 4, secondary address 0, with line feed as the message terminator, the following OPEN statement could be used:

```
Open #4:"GPIB0(pri=4,sec=0,eom=<10>):"
```

The PRI parameter specifies a primary address of 4, the SEC parameter specifies the secondary address, and the EOM parameter specifies the message terminator character.

Physical parameters, on the other hand, set interface characteristics, such as the baud rate of the COMM port. Physical parameters are set using the SET DRIVER statement. The following statement sets the baud rate of the COMM0 port to 4800 baud and selects even parity.

```
Set driver "COMM0(bau=4800,par=even):"
```

The ASK$ function returns parameters to the user or program that indicate the status of the device. To get information about a device, just specify the logical unit number for that device in an ASK$ function as shown below.

```
Print ask$("LU",4)
```

This statement prints the stream spec currently associated with logical unit number 4.

Interrupt Handling

4041 BASIC also provides a complete set of high-level commands for handling special conditions called "interrupts." An interrupt may occur as a result of:

- A specified condition on the GPIB including:
  - Receipt of a Device Clear (DCL) message.
  - End or Identify (EOI) line asserted.
  - Interface Clear (IFC) line asserted.
  - Receipt of the 4041's My Listen Address (MLA).
  - Receipt of the 4041's My Talk Address (MTA).
  - Service Request (SRQ) line asserted.
  - Receipt of the Take Control (TCT) message.

- Pressing the ABORT button on the front panel or program development keyboard or receiving a control-C from the COMM port.

- An error condition that occurs while executing a program.

- The completion of an I/O operation.

- Pressing a front-panel user-definable key or receiving the equivalent characters from the COMM port. (Control-F followed by a digit 0-9 is equivalent to pressing function keys 0-9 and control-D followed by a digit 0-9 is equivalent to pressing the corresponding digit on the keypad.)

Since all of these conditions are asynchronous (you usually don't know when they will occur in program execution), the 4041 allows you to set up a routine to handle any or all of these conditions and call the routine when the event occurs. For example, if a subprogram is set up to handle SRQs from the GPIB, you can tell the 4041 to temporarily stop what it is doing and execute this subprogram whenever an SRQ occurs. After the SRQ handling routine is complete, the 4041 returns to what it was doing when the interrupt occurred.

One particular advantage of this interrupt capability is in handling program errors. In a production environment it is especially important that the program be capable of recovering from errors in an orderly manner. The ON ERROR statement in 4041 BASIC allows you to handle error conditions in the program without aborting program operation. In addition, each subprogram can handle its own interrupts independently.

Section 5 discusses interrupt handling in more detail.

Debugging Programs

4041 BASIC includes some advanced features that make locating and fixing software problems simpler. Several special "debugging" commands are included. Table 3-4 lists the debugging commands and briefly describes each command.

TABLE 3-4
4041 BASIC DEBUGGING COMMANDS

Command	Description
BREAK	Sets breakpoints in the program at the specified line numbers.
CONNECT	Allows TRACE to work in local subprogram environments.
CONTINUE	Resumes execution of a program after it is halted by a breakpoint.
DEBUG	Runs the current program with breakpoints and trace flags enabled.
NOBREAK	Clears breakpoints in the current program.
NOTRACE	Clears trace flags set with the TRACE command.
SET DEBUG	Selects the device where debugging information is sent.
SET SYNTAX	Selects the device where syntax error messages are sent.
TRACE	Sets flags to display trace information on variables, branches in the program, and program execution.

Breakpoints. The 4041 allows you to temporarily halt the execution of a program to examine or modify a program line or the contents of a variable or to examing other aspects of program execution. The temporary halting points are called "breakpoints." Breakpoints are set with the BREAK command and are cleared with the NOBREAK command. Breakpoints do not add lines to the program and are completely ignored when the program is run with the RUN command. The breakpoints are enabled when the program is run

with the DEBUG command.

When a breakpoint is encountered, the PAUSE button is pressed, a control-B is received from the COMM port, or a STOP statement is executed, program execution stops. Execution can be resumed with the CONTINUE command or by pressing the CONTINUE key on the program development keyboard, or by pressing the PROCEED key on the front-panel keypad.

Tracing a program. In addition to the breakpoint feature, the 4041 can display information about the execution of a program. The TRACE command sets flags to:

- Print the line number of every program line as it is executed.
- Print a list of the name and contents of all variables or any specified variable each time they are modified in a program line.
- A list of the source and destination line number for all branches, including GOTO, GOSUB, CALL, user-defined funtion calls, or returns from subprograms or user-defined fucntions.

Complete information on the debugging features of the 4041 is provided in the 4041 Programmer's Reference Manual.

Section 4—Programming a 4041 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 system components, and a powerful programming language like 4041 BASIC, the job is greatly simplified.

This section provides a guide for writing 4041 BASIC programs to control a GPIB system. The details of interacting with GPIB devices, transferring data, and bus control are covered. A generous supply of sample programs is included to illustrate each point.

System Power-up

Power-up test. The first thing you'll notice when you power up your GPIB system is that most instruments go through some kind of a self-test procedure. During the self-test, the instrument usually won't respond to front-panel or GPIB input—all you can do is wait. The time required for the tests varies from a few milliseconds to several seconds, depending on the instrument.

If all goes well in the self-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.

Power-up SRQ. When the power-up test is complete (whether or not it detected errors), most instruments assert the SRQ (Service Request) line on the GPIB to tell the controller that the instrument status is ready to be read. By reading the status byte, the controller can tell if the instrument powered up normally or if errors were detected during self-test.

Handling SRQ interrupts is discussed in more detail in the next section. For now, you can assume that the instruments power up normally, so the power-up SRQ can be ignored. The 4041 powers up with SRQ interrupts disabled so unless you explicitly enable them, SRQs will be ignored. Later you'll probably want to enable SRQs so the program can detect power-up errors or other system conditions.

4041 GPIB Defaults

When the 4041 powers up, it assumes default values for a variety of parameters that affect GPIB communication. The defaults are set up for communicating with Tektronix GPIB instruments. The following paragraphs list the default parameters, describe the parameters, and show you how to set them to different values. With a system of Tektronix instruments, you may not have to change any of the defaults.

Default GPIB parameters. Table 4-1 shows the default parameter values. Notice that the parameters fall into two general catagories—physical parameters and logical parameters. (The difference between physical parameters and logical parameters is discussed in Section 3.) The default values are for the GPIB only. Some of these parameters also apply to other devices, but the default settings may be different for other devices.

The PRI (Primary Address) and TIM (Timeout) parameters have different default values when the Logical Unit Number (LUN) is explicitly opened with an OPEN statement and when the LUN is implicitly open (a LUN that is referenced without being explicitly OPENed). The primary address defaults to 31 for an explicitly open LUN and to the LUN number for LUNs that are implicitly opened.

The TIM parameter is set to 4 seconds when the LUN is implicitly opened and to infinity when the LUN is explicitly open.

Default LUNs. The 4041 assigns default stream specifications to LUNs 0-30 that correspond to primary addresses 0-30 on the standard GPIB port (GPIB0). These default stream specs use the default logical parameters shown in Table 4-1. As a result, the default LUNs can often be used without previously OPENing them. LUNs 31-32767 are assigned to the console device by default. These LUNs can also be assigned to other devices with an OPEN statement. Table 4-2 summarizes the default LUN assignments.

GPIB Parameters

The GPIB parameters are briefly described here. More information is provided in the 4041 Programmer's Reference Manual.

Physical parameters:

MA—My Address. This parameter sets the 4041's primary address. This is the address the

TABLE 4-1
DEFAULT GPIB PARAMETERS

Physical Parameters

Parameter	Valid Values	Default Value
MA	0-30	30
SC	YES/NO	YES
PNS	0-255	0
IST	TRUe/FALse	FALSE
DEL	NORmal/FASt	NOR
TC	SYN/ASY	SYN

Logical Parameters

Parameter	Valid Values	Default Value
PRI	0-31	LUN number (when LUN is implicitly open) 31 (when LUN is explicitly open)
SEC	0-32	32
EOA	<char>*	<44> (comma)
EOH	<char>	<32> (space)
EOM	<char>	<255> (carriage return/line feed)
EOQ	<char>	<0> (no output)
EOU	<char>	<59> (semicolon)
TIM	any positive number	4 seconds (when LUN is implicitly opened) infinity** (when LUN is explicitly open)
SPE	any positive number	10 milliseconds
TRA	NOR/FAS/DMA	NORmal

* <char> = any printing ASCII character or any ASCII decimal equivalent code enclosed in angle brackets.

** Actually, the default TIM parameter is 2.14748 E+7 or about 248 days.

TABLE 4-2
DEFAULT LUN ASSIGNMENTS

LUN	Default Assignment
0-30	GPIB devices with primary addresses 0-30
31-32767	Console Device

4041 uses for all GPIB communication whether the 4041 is the controller-in-charge or not.

SC—System Controller. If this parameter is set to YES, the 4041 is the system controller. When the parameter is set to NO, the 4041 relinquishes system control to another device. However, the 4041 can still become the controller-in-charge if the system controller or another controller passes control to it. (A discussion of the roles of the System Controller and the controller-in-charge is provided in Section 2.)

When the SC parameter is set to SC=YES, the 4041 asserts IFC (Interface Clear) for about 100 microseconds and asserts REN.

The 4041 powers up in a special "waiting" state. It does not assert control of the GPIB immediately. However, any bus activity initiated by the 4041 (such as executing a PRINT or INPUT statement), causes the 4041 to implicitly set SC=YES.

PNS—Polled with Nothing to Say. This parameter specifies the value the 4041 should report when it has nothing to say and it is polled by the controller-in-charge. Regardless of the value specified, bit 7 is cleared, since the 4041 is not asserting SRQ.

IST—Interface Status. This parameter sets a status flag that is used to report the 4041's status when it is parallel polled. The parameter has only two valid values—true or false.

DEL—GPIB settling time delay. This parameter sets the settling time for GPIB data byte transfers (called "T1" in the IEEE 488 standard). There are two possible settings—NORmal and FASt.

TC—Taking Control of the Bus. The TC parameter controls the 4041's ability to assert ATN. The 4041 asserts ATN whenever it is going to send an interface message. However, if the TC parameter is set for SYNchronous (the default), the 4041 must wait to assert ATN if a handshake cycle is in progress. The 4041 will wait up to TIM seconds for the handshake cycle to complete. If it does not complete within that time, a timeout error is issued. If the 4041 is set for TC=ASY, it can assert ATN at any time, regardless of the state of the bus. This means the 4041 can get control of the bus at any time. But, it also means that the 4041 could corrupt a data transfer that was in progress by asserting ATN during the transfer.

Setting physical parameters. Physical parameters are set with the SET DRIVER statement. A typical SET DRIVER statement is shown below.

```
Set driver "GPIB0(ma=16,sc=no):"
```

Physical parameters that are not specified in a SET DRIVER statement are unaffected—they retain their previous values.

Logical parameters:

PRI—Primary Address. This parameter specifies the primary address of a device on the bus. When the value of this parameter is between 0 and 30, the stream spec refers to a specific device with that address. A value of 31 addresses the GPIB I/O to the port itself, so all devices that are already addressed on that port participate in the transfer.

SEC—Secondary Address. This parameter specifies the secondary address of a device. The default value (32) causes no secondary address to be sent.

EOH—End Of Header. The End-Of-Header parameter specifies an ASCII character that is automatically inserted in GPIB output when the first semicolon is encountered in a PRINT list (not including semicolons in literal strings). The default EOH character is space, which is the character the Tek Standard Codes and Formats specifies to separate a command header from it's arguments.

The EOH character can be set to any printing ASCII character or any ASCII decimal code enclosed in angle brackets. If the value is set to zero, no character is output for EOH. The following examples illustrate valid EOH settings:

```
EOH=#
EOH=<9>
EOH=<0>
```

EOA—End Of Argument. The end-of-argument parameter specifies an ASCII character that is sent automatically whenever a semicolon is encountered in a PRINT list after the first semicolon (not including semicolons in literal strings). The default EOA character is comma, which Tek Codes and Formats specifies as the delimiter for successive arguments.

The valid values for the EOA parameter are the same as for the EOH parameter.

EOU—End Of Unit. This parameter specifies the character that will be sent as the end-of-message-unit delimiter. It is sent whenever a comma is encountered in a PRINT list and between elements of an array. The default EOU character for the GPIB is semicolon.

EOU can be set to the same values as EOH.

EOQ—End Of Query. The end of query character is automatically inserted at the end of a prompt message sent with an INPUT PROMPT. Valid settings are the same as EOH.

The default setting for EOQ is no output. For Tek Codes and Formats instruments, this parameter can be set to EOQ=?. Then the 4041 automatically inserts the question mark at the end of a prompt string. For example,in the following program segment, the question mark is

automatically added to the end of the prompt string (ID). The resulting query is ID?.

```
100    Open #1:"GPIB0(pri=1,eoq=?):"
110    Input #1 prompt "ID":ident$
```

EOM—End Of Message. This parameter specifies a character that is automatically appended to the end of a device-dependent message on output and that delimits a message on input. Valid settings for the EOM parameter are the same as for the EOH and EOA parameters with one difference. If EOM is set to 255, a carriage return and a line feed are appended to the end of the message and the EOI line is asserted concurrently with the line feed. If EOM is set to 0, no character is appended and EOI is asserted concurrently with the last byte in the message.

Figure 4-1 shows a typical PRINT statement and the resulting device-dependent message traffic (the addressing and unaddressing traffic have been omitted for clarity). The first command header (REC) is followed by the EOH character (space), the NUMREC value (1), the EOA character (comma), and the RECLEN value (1024). Next, the EOU character is sent to delimit the first message unit (REC 1,1024) from the following message unit (ARM A). At the end of the entire message, the EOM character is appended and EOI is asserted with the EOM character.

TIM—Data Transfer Timeout. The amount of time the 4041 will wait for each data byte to be transferred on the GPIB is set by this parameter. At power-up, all LUNs are assigned a timeout value of four seconds. When a LUN is OPENed, the timeout value defaults to infinity if the TIM parameter is not specified in the stream spec.

SPE—Serial Poll Timeout. This parameter sets the amount of time the 4041 will wait for a device to respond to a serial poll before going on to the next device or generating an error.

TRA—Data Transfer Mode. This parameter sets the GPIB data transfer mode. It can be set for NORmal, FASt, or DMA (with Option 1). The differences between these modes are discussed later in this section.

Setting logical parameters. Logical parameters are set when a logical unit number is opened with the OPEN statement. The OPEN statement performs two functions. First, it associates a specified logical unit number with a GPIB device or address. Second, it specifies the logical parameters that define the characteristics of the data transfer. A typical OPEN statement for a GPIB device is shown below:

```
Open #4:"GPIB0(pri=4,sec=2,eom=<10>):"
```

Logical parameters that are not specified in the OPEN statement are set to their default values.

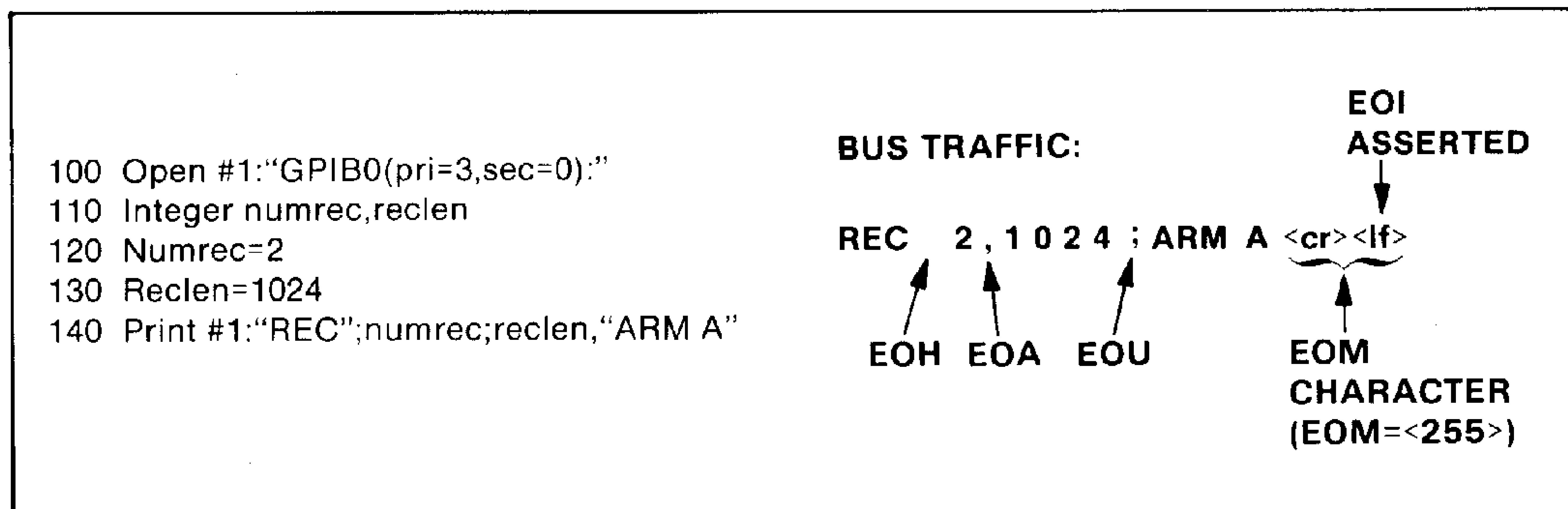


Fig. 4-1. *4041 BASIC automatically inserts the EOH, EOA, and EOM characters in a statement to build a valid message for a Tektronix Codes and Formats instrument.*

Talking to the Instruments

Once the system is configured and powered up, you're ready to begin communicating with the instruments. Remember from Section 1 that all messages transferred across the bus are divided into two types: interface messages and device-dependent messages. Interface messages control the operation of the bus and interfaces while device-dependent messages control instrument operations.

Device-dependent messages. Device-dependent messages comprise 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. However, the Tektronix Codes and Formats standard specifies the content and syntax of device-dependent messages for Tektronix GPIB instruments.

Tektronix instruments implement two general types of commands: set commands and query commands. Set commands are device-dependent messages that set instrument operating modes or control device operations. Query commands are device-dependent messages that return current instrument settings, status, or values. Many query commands are simply set commands with a question mark added to the end of the command. For example, the VPOS command sets the positive supply output voltage for a PS 5010 Programmable Power Supply. The VPOS? query returns the positive supply voltage setting.

Transferring a device-dependent message. Though the IEEE 488 standard does not specify the content of the device-dependent messages, it does specify a protocol for how device-dependent messages should be transferred over the GPIB. There are three basic parts to every device-dependent message transfer (see Fig. 4-2).

1. The addressing sequence. The controller addresses one instrument (or itself) as a talker and one or more instruments (including itself) as

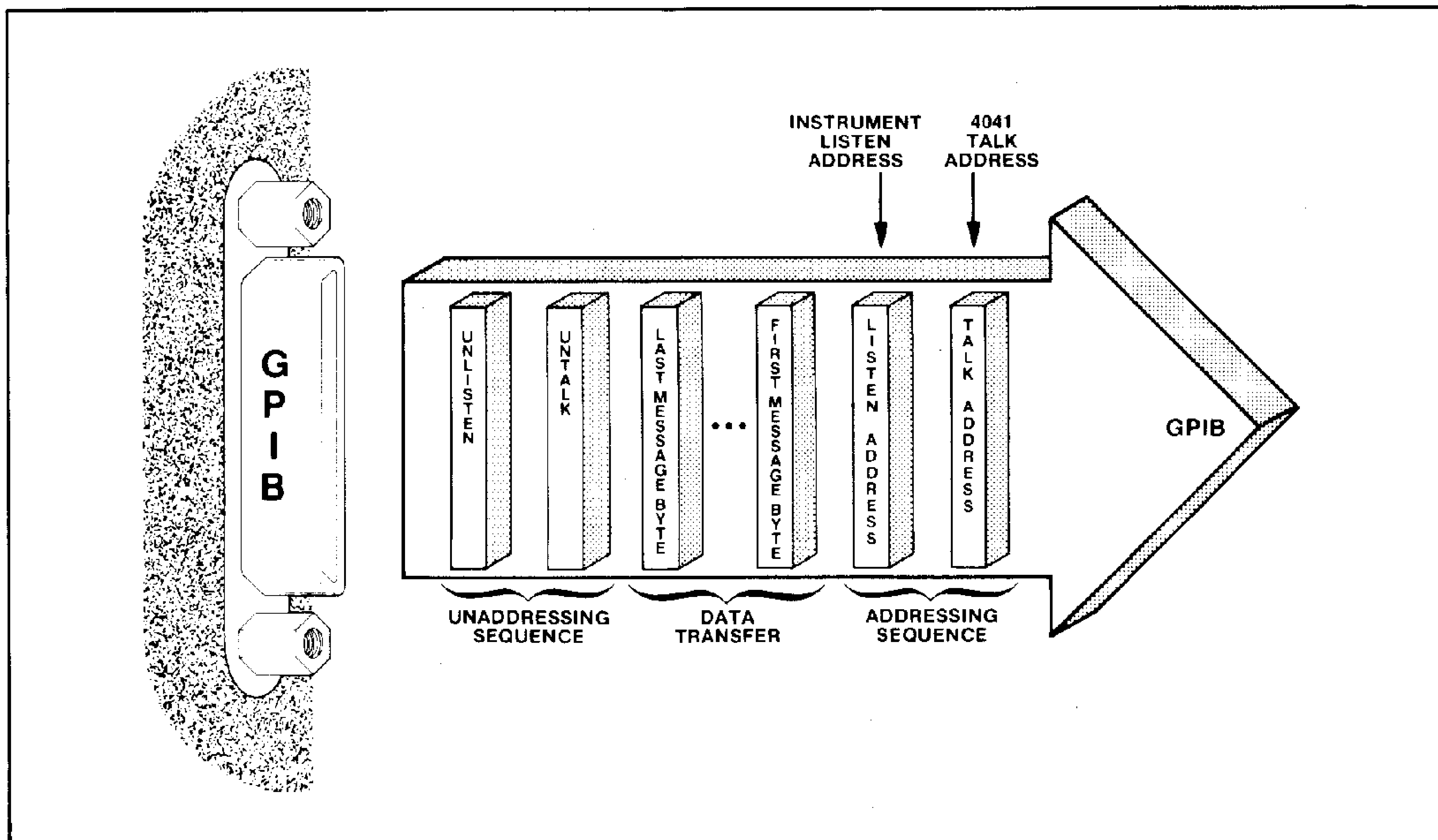


Fig. 4-2. *There are three basic parts to every data transfer—the addressing sequence, the data transfer, and the unaddressing sequence.*

listeners. If the 4041 is sending data, such as with a PRINT statement, it sends the instrument's listen address (primary address+32) and its own talk address (MA+64). If the 4041 is receiving data, such as with an INPUT statement, it sends the instrument's talk address (primary address+64) and its own listen address (MA+32) as shown in Fig. 4-3. If the instrument requires a secondary address, this address follows the primary address.

2. The data transfer. When the addressing sequence is complete, the talker begins sending the device-dependent message to the listeners. The talker indicates the end of the message by asserting EOI with the last character of the message or by adding a line-feed character to the end of the message.

3. The unaddressing sequence. Most controllers send the UNTalk and UNListen interface messages at the end of a transfer to unaddress the current talkers and listeners. This insures that the bus is clear and ready for the next transfer.

To see how this works, consider sending an "ARM A" command from the 4041 to a Tektronix 7612D Programmable Digitizer. For the sake of this example, assume that the instrument is set for primary address 3 and secondary address 0. The bus traffic required to send this message is shown in Fig. 4-4.

First, the controller asserts the ATN line and sends its own talk address, followed by the instrument's primary listen address (3+32=35). This address tells the instrument that it should listen to the message that follows. Next, the controller sends the mainframe secondary address (0+96=96). In the case of the 7612D, the secondary address tells the mainframe to listen instead of one of the plug-ins installed in the mainframe.

With the addressing sequence complete, the 4041 unasserts ATN, and begins transferring the device-dependent message (ARM A). The message is transferred one character at a time. If an EOM character is defined (EOM is not set to 0), the EOM character is sent at the end of the message. With the last character (EOM or the last character of the message), the 4041 asserts EOI to tell the 7612D that this is the end of the message.

When the ARM A message has been transferred, the 4041 asserts ATN again and sends the universal UNTalk and UNListen interface messages.

This process is the same regardless of the content and format of the message. In some cases, the secondary address may be omitted if the instrument does not require a secondary address.

PRINTing an ASCII message. The process of transferring a message may seem a bit complex, but the 4041 takes care of most of the details automatically. All you have to do is use a PRINT or INPUT statement with the proper stream spec or logical unit number. (Section 3 discusses stream specs and logical unit numbers in more detail.)

To make communicating with a Tektronix instrument even simpler, Logical Unit Numbers

PRIMARY ADDRESS	LISTEN ADDRESS	TALK ADDRESS
Ø1	32	64
1	33	65
2	34	66
.	.	.
.	.	.
.	.	.
3Ø	62	94

Fig. 4-3. *The 4041 automatically generates the listen address on PRINT statements by adding 32 to the instrument's primary address. On INPUT statements, the talk address is generated by adding 64 to the instrument's primary address. The 4041's talk or listen address is generated by adding the appropriate offset to the MA parameter value.*

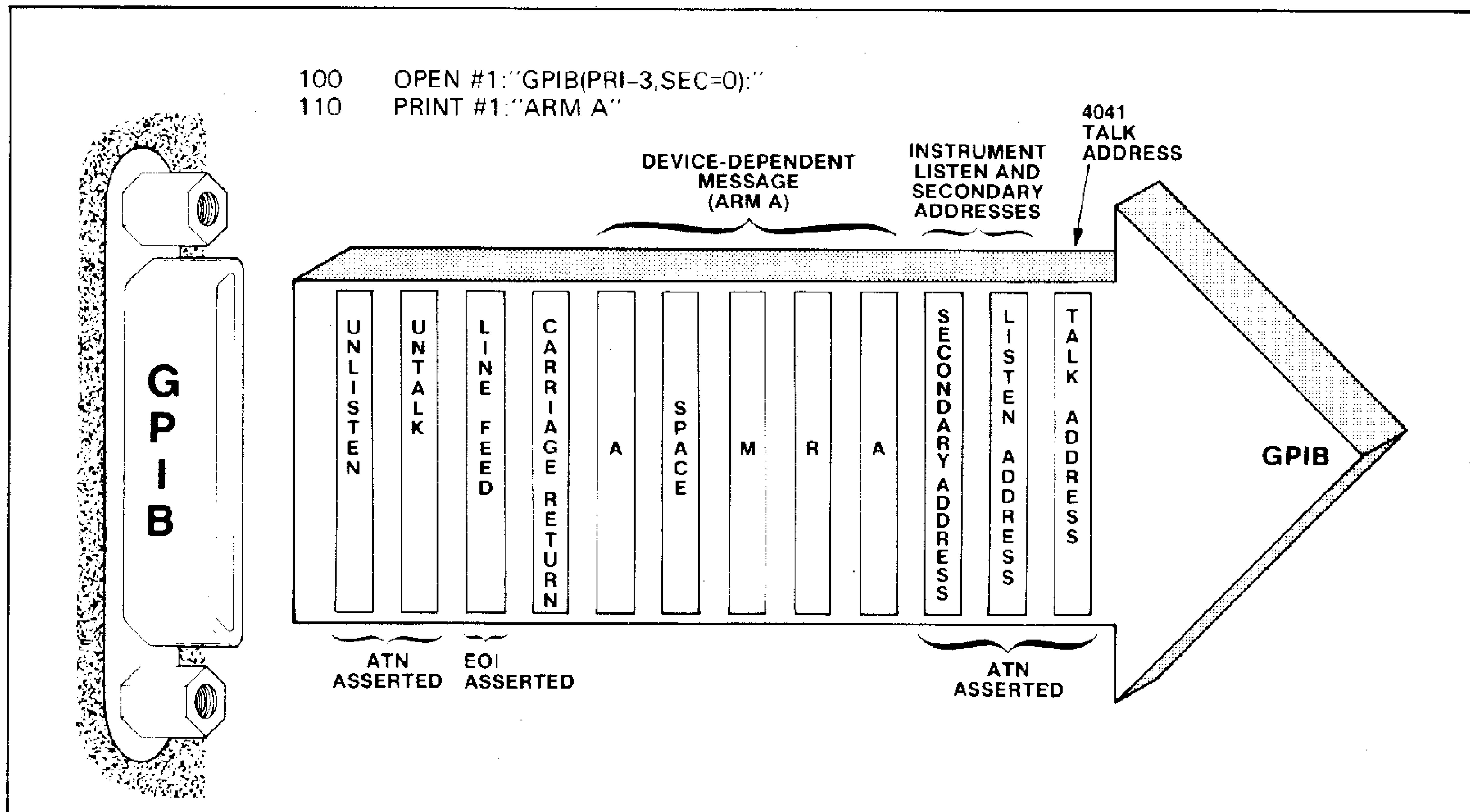


Fig. 4-4. *A typical PRINT statement sending the device-dependent message "ARM A" and the resulting message traffic.*

0–30 are automatically assigned default stream specs that correspond to primary addresses 0–30. The default stream specs are set to handle instruments that implement the Tektronix Codes and Formats Standard and that require no secondary addresses.

These default LUNs allow you to begin communicating with most Tektronix instruments immediately. For example, the following PRINT statement sends a SET? query to a DM 5010 Programmable Digital Multimeter set for address 16 using the default stream spec for LUN 16.

```
Print #16:"SET?"
```

If your instrument requires secondary addresses or some other non-default stream spec, you simply OPEN the logical unit with the required stream spec parameters. To send the same SET? query to a 7612D Programmable Digitizer set for primary address 4 and secondary address 2, the following statements could be used:

```
100    Open #10:"GPIB0(pri=4,sec=2):"
110    Print #10:"SET?"
```

The PRINT statement can be used to send any ASCII message. The message can be included as a literal string in quotes, as illustrated above. Or, the message can be stored in a variable and the variable can be included in the PRINT statement like this:

```
Command$="SET?"
      :
Print #10:command$
```

Numeric values can be stored in numeric variables and included in a PRINT statement to a GPIB device. The PRINT statement automatically converts the numeric value to a series of ASCII digits. Numeric values and character strings can be mixed freely in a PRINT statement as long as the result is a valid command string for the instrument.

In the following program segment, the output voltage for a PS 5010 Programmable Power Supply is stored in the numeric variable, POSVOLTS. The complete message for the PS 5010 is constructed from a literal command header (VPOS) and the numeric value stored in POSVOLTS. Notice that the 4041 automatically inserts the EOH character

(space) between the header (VPOS) and the argument (3.5).

```
100     Posvolts=3.5
110     Print #22:"VPOS";posvolts
```

Messages can also be built dynamically based on user input. The following program segment asks the user to enter a desired output voltage for a PS 5010 Programmable Power Supply. The program accepts the input value from the system console and builds a message for the PS 5010 that tells it to set its output to the desired voltage.

```
100     Input prompt "Enter the output voltage: ":posvolts
110     Print #22:"VPOS";posvolts
```

Getting a response. The process of sending a query is the same as sending any other device-dependent message except that after the query is sent, the instrument expects to be addressed to talk so that it can transmit the query response. The complete process is shown in Fig. 4-5.

1. Address the instrument to listen. The 4041 sends the instrument's listen address and, if necessary, its secondary address. The 4041 addresses itself to talk by sending its talk address.

2. Send the query command. Next, the query command is sent to the instrument.

3. Unaddress the instrument. The instrument is unaddressed to prepare for getting the response.

4. Address the instrument to talk. The controller sends the instrument's talk address and, if necessary, the secondary address. The 4041 also addresses itself to listen by sending its listen address.

5. Get the response. The instrument sends the query response to the controller, terminating the message with EOI or EOI and line feed, depending on the message terminator setting.

6. Unaddress the instrument. The UNTalk and UNListen interface messages are sent again to clear the bus.

Query commands are sent with a PRINT statement like any other device-dependent message. The response from most queries can be read with an INPUT statement as shown below.

```
100     Print #22:"ID?"
110     Input #22:ident$
```

This program segment asks the instrument at address 22 to send an ID string that identifies the instrument type and firmware version number. Line 100 sends the query and line 110 reads the response string into the string variable IDENT$.

4041 BASIC provides an easier way of handling queries with a special INPUT clause called PROMPT. The INPUT PROMPT statement allows you to send the query and get the reply in a single program line. The program shown above can be shortened to a single line with the INPUT PROMPT statement:

```
100     Input #22 prompt "ID?":ident$
```

This single program line performs the same function as the separate PRINT and INPUT statements, but it's more efficient because the query is completed in a single program line.

Receiving long strings. String variables in 4041 BASIC can contain up to 72 characters by default. If the query response is longer than 72 characters, the string variable must be dimensioned to at least the length of the query response string.

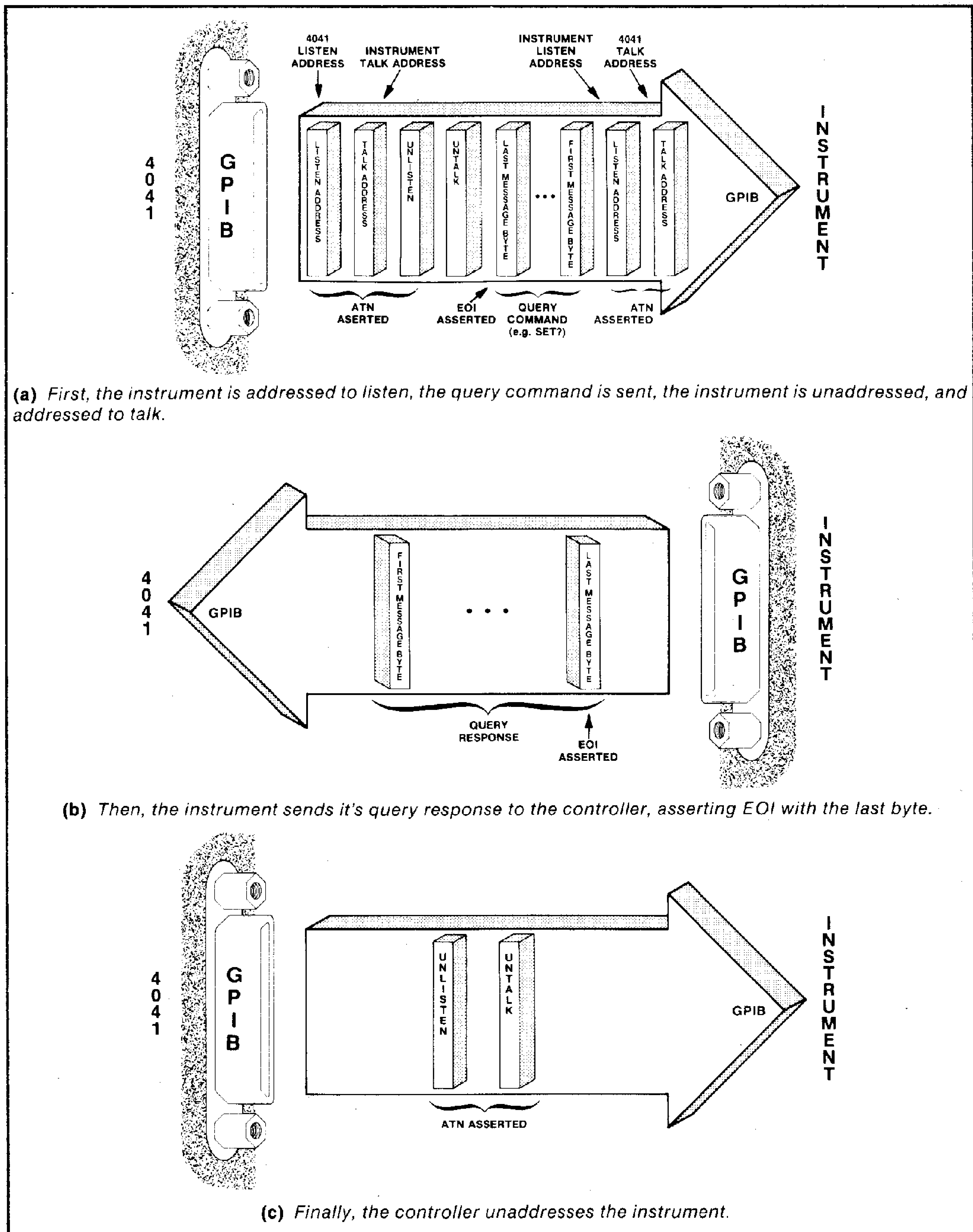
For example, the program segment below gets all the current settings of a PS 5010 Programmable Power Supply using the SET? query. The response from the SET? query is 127 characters long, so the string variable, SETTING$, must be dimensioned to at least 127 characters in length.

```
100     Dim setting$ to 150
110     Input #22 prompt "SET?":setting$
```

A string variable can be dimensioned to longer than the expected response string. The string input will be terminated as usual and unused space in the string variable is ignored.

Notice that the DIM statement can dimension the length of individual strings as well as set up string arrays. If the keyword "TO" is included as in the previous example, the string is dimensioned to the specified length. If the dimension parameter is enclosed in parentheses directly following the variable name, a string array is set up. The following dimension statement illustrates the two functions of the DIM statement.

```
Dim setting$(32) to 128
```



(a) *First, the instrument is addressed to listen, the query command is sent, the instrument is unaddressed, and addressed to talk.*

(b) *Then, the instrument sends it's query response to the controller, asserting EOI with the last byte.*

(c) *Finally, the controller unaddresses the instrument.*

Fig. 4-5. *The process of sending a query and getting the response.*

This statement sets up a string array consisting of 32 strings. Each string in the array can contain up to 128 characters.

Inputting numeric values. Some instrument queries return numeric values. For example, the LEV? query for a 7612D Programmable Digitizer returns the current trigger level. The query response looks like this:

```
LEV 114;
```

This response means that the current trigger level setting is 114. The response could be read into a string variable as previously described. However, if you want to use the numeric part of the query response in a computation, the numeric part must be extracted from the string with the SEG$ function and the value converted to a numeric variable with the VAL function.

A simpler way to get the numeric value is to use a numeric variable in the INPUT statement that reads the query response. The 4041 reads the query response and throws away all the non-numeric characters (except plus or minus signs and "E" when used in scientific notation). The numeric part of the response is stored in the specified variable.

If the query response contains more than one numeric value, one variable must be specified for each value. Consider, for instance, the REC? query for a 7612D Programmable Digitizer. The response to this query indicates the current number and length of waveform storage units called "records." The response looks like this:

```
REC 2,1024
```

The response says that there are two records of 1024 points each currently set-up in the selected 7612D channel. To read this query response, two numeric variables must be specified—one for the number of records value and one for the record length value.

```
100    Open #10:"GPIB0(pri=10,sec=4):"
110    Input #10 prompt "REC?":numrec,reclen
```

Transferring Waveforms

Some instruments may send or receive larger blocks of data than simple set or query commands. Probably the most common example of a large data block is waveform data from a digitizer. However, this type of data is not limited to waveforms only. For example, some TM 5000-series instruments can send a block of data that represents all instrument settings in "low-level" form.

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 don't have any choice which format to use because the instrument requires one format. Other instruments, like the 7D20 Programmable Digitizer, can transmit or receive data in either format.

The following paragraphs describe transferring both waveform formats. Though the discussion is particularly oriented to waveform data, any type of array or block data can be handled in a similar fashion.

Tek Codes and Formats waveforms. The Tektronix Standard Codes and Formats specifies two general parts of a waveform—the preamble and the data itself. The preamble contains information about the waveform data, such as the data encoding (ASCII or binary), the number of waveform points, the scale factors, etc. This preamble information may be sent with the waveform or it may be sent separately in response to a WFMPRE? query. A typical waveform preamble sent by the 7D20 Progammable Digitizer is shown in Fig. 4-6. Appendix B describes the contents of the waveform preamble in more detail.

The contents of the preamble varies with different instruments and data encoding, but the format of each parameter included in the preamble is specified. For example, the data encoding is always specified as ENCDG:ASCII or ENCDG:BINARY, and the number of data points is always specified as NR.PT:nnnn (where nnnn is the number of data points). This consistency makes handling a waveform from any instrument that implements the Tek Standard Codes and Formats simpler.

Waveform data can be transmitted in one of three basic formats—ASCII, binary block, or end binary block. The ASCII format is shown below:

```
CURVE <data point>,<data point>,<data point>...
```

This is the simplest data format. It can be sent and received with standard PRINT and INPUT statements. Each data point is simply an ASCII-

WFMPRE WFID:W 1,ENCDG:ASCII,NR.PT:1024,PT.FMT:Y,XINCR:1.0E-5,
PT.OFF:1.2E+1,XZERO:0,XUNIT:S,YMULT:1.0,YZERO:0,YUNIT:V

Fig. 4-6. *A typical waveform preamble from the 7D20 Programmable Digitizer.*

coded number. They may be integers, floating point values, or in scientific notation.

The second data format is called binary block. It is slightly more complex than ASCII and requires some special handling. However, it also has two significant advantages. First, the binary block format contains some error checking that the ASCII format does not have. This error checking provides a means of detecting data transmission errors. Second, a waveform can usually be transferred much faster in binary block format than in ASCII format since each binary data point can be transmitted in one or two bytes. In ASCII, each data point requires one byte per character in the number including the comma and, if required, the minus sign. In many cases, each data point may require six or more bytes to transmit.

The binary block format is:

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

CURVE is an ASCII header followed by a space. This header may be omitted by some instruments.

% is the ASCII code for a "%" character. This identifies the data that follows as a binary block.

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 sends more than 8-bits of data per data point, two bytes are used per point with the most significant byte sent 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 (%). In other words, it's the eight-bit sum of the preceding bytes, ignoring the carry. If the listener 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.

The third data format is a simpler form of the binary block. It is called the "end block binary" format. Data points are encoded in one or two binary bytes as in the binary block format. However, the byte count and checksum are omitted. To differentiate between the two formats, the first character is changed to an at sign (@). The format is:

CURVE @<data value><data value>...<data value>

CURVE is an ASCII header followed by a space. This header may be omitted by some instruments.

@ is the ASCII code for the "@" character. This character identifies the following data as end block binary format.

DATA VALUE is an 8-bit binary number as in the binary block format. EOI is asserted with the last data byte.

Receiving ASCII data. INPUT can be used to accept ASCII waveform data in a manner similar to using INPUT to accept query responses. The only difference is that a numeric array variable is specified instead of a string or simple numeric variable. The numeric array must be dimensioned to the number of points sent by the instrument. The number of points can be determined from the waveform preamble.

For example, the program shown in Fig. 4-7 gets an ASCII waveform from a 5223 Digitizing Oscilloscope. The CURVE header that preceeds the waveform data is ignored since the variable in the

INPUT statement is a numeric array.

```
100    Input #6 prompt "WFMPRE?":wfmpre$
110    Nrpt=valc(wfmpre$,pos(wfmpre$,"NR.PT",1))
120    Dim waveform(nrpt)
130    Input #6 prompt "CURVE?":waveform
```

Fig. 4-7. *Getting an ASCII waveform from a 5223 Digitizing Oscillosope.*

Line 100 gets the number of points by reading the waveform preamble. The number of points parameter is extracted from the response string in line 110 and this value is used to dimension the waveform array variable in line 120. Finally, the waveform data is read using an INPUT statement with the PROMPT clause.

The waveform can also be stored in a string variable. Storing the data in a string variable is a little faster, but the data is difficult to process in that form. (If it is necessary to process numeric data stored in a string variable, you can use the GETMEM statement to translate the data into numeric format. GETMEM is described later in this section.) However, if the waveform is only going to be stored on tape for later retrieval, the string variable may be the best approach. Figure 4-8 shows a program to get a waveform and store it in a string variable.

This program is the same as the one in Fig. 4-7, except that the entire waveform response string is stored in WAVEFM$. The string is not translated into numeric variables. Each sample contains three to four characters, sometimes a minus sign, and a comma to delimit the samples. In addition, the header contains several characters. The length of the string variable must be dimensioned to hold all of these characters. In this example, the string variable is dimensioned to five times the number of samples plus 6 (the header contains 6 characters).

Sending ASCII data. The PRINT statement can send waveform data from a numeric array or string variable. Remember that the proper command must be sent ahead of the waveform data. For example, the 5223 Digitizing Oscilloscope expects waveforms to be sent in the form:

```
CURVE 113,104,98,87,...,-231,-247,-313
```

If the data is stored in a numeric array, the header (CURVE) must be sent before the waveform data as shown below. The second print statement shows how a waveform stored in a string variable is sent. In this example, the string contains the CURVE header as it would be if the waveform was previously read from an instrument.

```
Print #10:"CURVE";waveform
Print #10:wavefrm$
```

When the 4041 is transmitting a numeric array, such as in the first PRINT statement, it automatically sends the EOU character between each array element as a delimiter. At the end of the message, the EOM character is sent. Data stored in a string variable is sent exactly as it is stored, except that the EOM character is added to the end of the message.

Receiving binary data. Binary waveform transfers are handled a little differently than ASCII waveform transfers. The INPUT statement is still used to accept data, but the 4041 must be told that the data is being transferred in binary rather than the default ASCII format for INPUT.

The USING clause for the INPUT statement allows you to define the format of the input data. You can specify exactly what type of data and how much data you expect the INPUT statement to

```
100    Input #6 prompt "WFMPRE?":wfmpre$
110    Nrpt=val(seg$(wfmpre$,pos(wfmpre$,"NR.PT",1)+6,4))
120    Dim wavefm$ to nrpt*5+6
130    Input #6 prompt "CURVE?":wavefm$
```

Fig. 4-8. *Reading an ASCII waveform into a string variable.*

accept. The type of data is specified with a series of "operators." A variety of operators are provided for string input, numeric input in several formats and radices, and for binary data input. This section describes only the two operators that are designed specifically for receiving binary block and end block data.

Block binary data input. The "%" operator tells the 4041 that the data to be input is in binary block format. The 4041 automatically reads the leading percent sign, the byte count, the data bytes, and the checksum. The 4041 computes a checksum of the block and adds the checksum sent by the instrument. If the result is zero, the transmission is correct. If not, the 4041 reports an error.

A typical INPUT USING statement for reading a binary block is shown below.

```
Input #1 using "+8%":waveform
```

The numeric modifier (+8) that precedes the "%" operator specifies how the binary data is stored in the array (WAVEFORM). If the value of the modifier (call it "n" for this discussion) is less than or equal to 8, the 4041 stores the left-most n bits of each data byte received from the bus as the right-most n bits in a 4041 integer. If the value of n is less than 16 bits but greater than 8, the 4041 stores the n left-most bits of each **pair** of bytes as the right-most n bits in a 4041 integer. If the value of n is 16 (the default value), each byte pair is stored directly as a 4041 integer.

Notice that if the value of the modifier is less than or equal to 8, the 4041 reads one byte per variable or array element. If the value of n is greater than 8 and less than or equal to 16, the 4041 reads two bytes per variable or array element. If the array is dimensioned larger than the amount of data sent by the instrument, the array is filled to the point where the instrument finished talking. The remainder of the array is not affected. If the array is smaller than required for the data, the 4041 stops reading data at the end of the array.

A plus or minus sign modifier may also preceed the numeric modifier. If no sign is specified or if a minus sign is specified, the received data is assumed to be in two's complement notation. If a plus sign is specified, the value is considered a positive integer. All unused bits in the 4041 integer value are set to zero.

When using the % or @ operators to transfer binary data, the waveform data must be stored in an integer array. After it is input, the data can be copied to a floating-point array if necessary.

Figure 4-9 shows a typical program segment that reads a binary block from a 7D20 Programmable Digitizer. The 7D20 can send waveform data in either binary or ASCII format. In this case, the DATA ENC:BIN command included in the prompt tells the 7D20 to encode output data in binary. The response to the CURVE? query in this mode is a CURVE header followed by a standard binary block of data. The program sends the query and gets the response in a single INPUT PROMPT statement with a USING clause. The first operator in the USING clause (6a) gets the five character header and the trailing space and puts it in HEADER$. The second operator in the USING clause (+8%) gets the binary block data and stores it in the integer array WAVEFORM.

Receiving end block binary data. The "@" operator accepts binary data in end block format. The "@" operator is used just like the "%" operator. The same numeric modifiers are used and data is handled in the same fashion. The only difference is that the end block format does not contain the byte count or checksum bytes, so no error checking is

```
100    Integer waveform(1024)
110    Input #1 prompt "DATA ENC:BIN;CURVE?" using "6a,+8%":header$,waveform
```

Fig. 4-9. *Binary block data can be read with a single 4041 statement. The 4041 automatically handles the binary block format, computing checksums and storing data in an integer array.*

performed. The INPUT statement does, however, check for the proper header character for end block format (@). An error is generated if the correct character is not received.

A typical INPUT statement for receiving end block binary data is shown below:

```
Input #1 using "+8@":waveform
```

Reading binary data into a string variable. When data transfer speed is critical, binary data can be read into a string variable. Since no translation is required for storing data in a string variable, the input is considerably faster than inputting to a numeric array. However, there are some special considerations for handling binary data.

First, since the binary data may contain valid ASCII characters, such as carriage return, the EOM character must be set to null (EOM=0). This setting makes the EOI bus signal the only valid terminator—no binary value will cause the INPUT to terminate.

Second, remember that the string will store data exactly as it is received. As a result, the header, byte count, and checksum will be included with the string. The data can be stored on tape in the string format for later processing or transmission back to the instrument. The GETMEM statement can also be used to translate the binary block or end block data to a numeric variable. An example of reading binary data into a string variable and using GETMEM to translate it is provided in the discussion of the GETMEM command later in this section.

Sending binary data. The USING clause can also be used with the PRINT statement to send binary block and end block binary data. The "%" and "@" operators are used with the PRINT USING statement in the same way as for INPUT USING. Two typical PRINT statements are shown below. The first one transmits a binary block to an an instrument. The second statement transmits the same data in end block binary format.

```
Print using "+8%":waveform
Print using "+8@":waveform
```

The PRINT statement automatically adds the proper header character ("%" or "@"), the byte count, and the checksum to the data. Data is transmitted on the bus according to the value of the numeric modifier as discussed for input. If the value of the modifier is greater than 8, two bytes are sent per value. If the value of the modifier is less than or equal to 8, one byte is sent per value.

When two bytes are transmitted, the most significant byte is transmitted first. Data is left justified in both bytes. If a plus sign modifier is included, the data is transmitted as integers with unused bits set to zero. Otherwise, data is transmitted in two's complement notation.

The source array must be an integer array. If any value in the array is larger than can be expressed in the number of bits specified with the "@" or "%" operators, the 4041 reports an error. For the "+8%" operator specified in the above examples, the largest value that can be transmitted in an 8-bit integer is 255. If the array contains a value larger than 255, the 4041 reports an error.

The program statement shown below sends a binary block waveform to a 492P Programmable Spectrum Analyzer. The header is sent first, followed by the binary block. The 4041 automatically computes the byte count and checksum and adds them to the data to make a complete binary block.

```
100     Print #1 using "6a,+8%":"CURVE ";waveform
```

Special I/O Situations

Suppressing the EOM character on PRINT. Sometimes you may need to split a message into two pieces and send each piece with a separate PRINT statement. One example of this is in sending binary waveform data with a PRINT USING statement.

The key to making this work is suppressing the message terminator that PRINT automatically sends at the end of a message. If a semicolon is appended to the end of the PRINT list, the 4041 suppresses the EOM character and EOI that are automatically inserted at the end of the message. Thus, when the second PRINT statement is executed, the instrument is still waiting for the remainder of the message. The second PRINT statement supplies the remainder of the message. Using this technique, the message can be broken into as many parts as required. Just append a semicolon to the end of each PRINT statement except the last.

Be careful not to confuse the semicolon included

in a PRINT list with the semicolon operator used in a PRINT USING statement. The semicolon operator in a PRINT USING or IMAGE statement sends a semicolon character on the bus. The semicolon character included in a list of variables for the PRINT statement suppresses the EOA, EOU, or EOM characters that are normally sent between variables in a PRINT statement.

Using alternate delimiters on INPUT. The default delimiters for numeric data input are space, tab, comma, semicolon, colon, and the EOM character. For string input, the default string terminator is the EOM character. In some cases, it is useful to define another delimiter to aid in breaking a message into smaller parts.

Two clauses for the INPUT statement allow you to temporarily define alternate delimiter characters. The DELN clause defines an alternate delimiter for numeric data and the DELS clause defines an alternate delimiter for strings. The DELS and DELN clauses do not disable the default delimiters.

As an example of using the DELS clause, consider the process of receiving a waveform from a 7854 Programmable Oscilloscope. The 7854 sends a waveform preamble ahead of the waveform data. The preamble contains scaling information, the number of data points and other information about the waveform. Figure 4-10 shows a typical 7854 waveform including the preamble and data.

The preamble information is usually best stored in a string or numeric variable separately from the waveform data. If the preamble is stored in a string variable, the string input must be terminated at the end of the preamble before the waveform data. There are three ways to accomplish this. First, the string can be dimensioned to the exact length of the preamble string. This is often difficult, since a change in one of the values included in the preamble can change the length of the preamble string. Second, the EOM character can be temporarily set to delimit on the semicolon following the preamble. This requires re-opening the LUN with a different EOM parameter after the preamble is transferred.

The third and simplest method is to use the DELS clause on INPUT. The DELS clause defines another delimiter other than EOM. Figure 4-11 shows how DELS can be used to read a preamble and waveform from the 7854.

```
100    Dim wfmpre$ to 100
110    Print #1:"0 WFM SENDX"
120    Wait 0.1
130    Input #1 dels chr$(13):wfmpre$,waveform
```

(a) *The DELS clause can be used to separate two parts of an INPUT message as long as both parts are INPUT in the same statement.*

```
100    Open #1:"GPIB0(pri=1,eom=<13>):"
110    Print #1:"0 WFM SENDX"
120    Wait 0.1
130    Input #1:wfmpre$
140    Input #1:waveform
```

(b) *The same thing can be accomplished by setting the EOM character to carriage return. Then, the two parts of the statement can be input in a single INPUT or in two separate INPUT statements.*

Fig. 4-11. *Input can be delimited either with the DELS clause or with the EOM character. Here, the preamble and waveform data from a 7854 are accepted using the DELS and EOM techniques.*

```
WFMPRE ENCDG:ASC,NR.PT:512,PT.FMT:Y,XZERO:0,XINCR:390.6E-09,
XUNIT:S,YZERO:0,YMULT:5,YUNIT:V;
CURVE 0.0586,0.0592,0.0684,0.0722,...,1.2426
```

Fig. 4-10. *The 7854 sends its waveform preamble and its waveform data in the same message. The preamble is delimited from the waveform data by a semicolon and a carriage return. These characters can be used to separate the preamble and data on input to the 4041.*

Part **a** of the figure shows the process of transferring the preamble and waveform using the DELS clause. Part **b** shows the same process using the EOM character. It is important to remember that the DELS clause does not delimit input the same way that the EOM character does. Input from the bus is actually halted when the EOM character is received. Input from the bus is NOT halted when the DELS character is received. The DELS clause only affects the way the 4041 stores data in the variables.

As a result, if one INPUT statement is terminated with a DELS or DELN and the remainder of the data is accepted with a separate INPUT statement, some data after the DELS or DELN character may be lost. If the DELS or DELN clauses are used to delimit individual variables in a single INPUT statement, no data is lost.

Another way to handle this situation is to use the BUFFER clause. After the first string input is terminated the ASK$("BUFFER",n) function returns the location where the string input was terminated. The remainder of the buffer contents can be stored with GETMEM. The BUFFER clause and GETMEM statements are described in more detail later in this section.

Proceed Mode

The 4041 has a special I/O mode called "Proceed Mode." In proceed mode, the 4041 can do just what the name implies—it can proceed with other processing while an I/O operation is in progress. This can significantly improve system performance by overlapping the time required for I/O with other processing.

To set the 4041 to proceed mode, the following statement must be executed:

```
Set proceed 1
```

This statement can either be executed in immediate mode or it can be entered as a line in a 4041 program. Proceed mode is disabled with a SET PROCEED 0 statement.

When two or more I/O statements are executed using the same output port (GPIB0, TAPE, etc.), the 4041 automatically delays execution of the I/O until all previous I/O on that device is complete.

When the 4041 is set for proceed mode, a few special restrictions apply to the PRINT and INPUT statement. These restrictions are described in the following paragraphs.

Proceed mode PRINT. There are no special restrictions on the amount or type of data for proceed mode PRINT statements. However, the 4041 does not proceed to the next statement after the PRINT until the data to be sent can fit in the I/O buffer. If the default I/O buffer is used (no BUFFER clause is specified), the 4041 proceeds to the next statement when there are 514 or fewer bytes to output. (The I/O buffer and BUFFER clauses are described later in this section.)

Figure 4-12 illustrates the use of proceed mode PRINT. In this example, a waveform of 1024 ASCII points is being transmitted. Each point contains 5 characters (four digits plus a comma delimiter) for a total of 5120 bytes. In part **a** of the figure, the 4041 does not continue until the first 4626 characters (9 buffers of 514 points each) have been sent. Then, when the last 514 bytes have been loaded in the output buffer, the 4041 proceeds to the next statement.

Once the last data from the array variable (WAVEFORM) is moved into the I/O buffer, the WAVEFORM variable can be modified. For example, new data could be stored in WAVEFORM while the old data is being sent.

```
100    Set proceed 1
110    Dim waveform(1024)
120    Print #1:"CURVE ";waveform
         .
         .
```

(a) *Using the default (514-byte) I/O buffer, execution does not proceed until the last 514-byte block is ready for transmission.*

```
100    Set proceed 1
110    Dim waveform(1024),buffer$ to 5124
120    Print #1 buffer buffer$:"CURVE ";waveform
         .
         .
```

(b) *When the BUFFER clause is used, a larger I/O buffer can be defined that will hold the entire data transfer. As a result, execution continues as soon as the I/O begins.*

Fig. 4-12. *The 4041 does not continue execution in proceed mode until the data to be transmitted can fit in the I/O buffer. If the default I/O buffer is used, this means that execution does not proceed until the last 514-byte block is ready to transmit.*

In part **b** of the figure, a 5124-byte BUFFER is specified, so the 4041 proceeds as soon as the data is loaded into the buffer string.

Proceed mode INPUT. Proceed mode INPUT is limited to one string variable per INPUT statement. No numeric input is allowed. The string can be used as an I/O buffer, storing data directly as it is received. The GETMEM statement or VAL, VALC, and SEG$ functions can be used to extract numeric values or substrings from the buffer string.

Though only one string variable can be used in a proceed mode INPUT statement, the string variable can be dimensioned to up to 32,767 characters.

INPUT statements in proceed mode may not use any clauses such as BUFFER or PROMPT except the # clause. The # clause specifies the LUN for the I/O.

Variables read with a proceed mode input cannot be referenced until the input is complete. If an attempt is made to reference a variable before the input is complete, execution is automatically delayed at the statement that refers to the input variable. Execution continues with the statement that refers to the variable when the input is complete (Fig. 4-13).

```
100    Set proceed 1
110    Dim setting$ to 200
120    Print #24:"SET?"
130    Input #24:setting$
         .
         .
200    Offset=valc(setting$,pos(setting$,"OFFS",1))
```

Fig. 4-13. *Execution is automatically delayed if a program line references a variable that is being filled by a proceed mode INPUT statement.*

In this example, a SET? query is sent to a FG 5010 Programmable Function Generator. The response is input in proceed mode to SETTING$. Line 200 extracts the DC Offset value from the setting string.

As soon as the input in line 130 is started, execution continues. However, if line 200 is reached before the input operation is complete, execution is automatically delayed. If execution was not delayed, the contents of SETTING$ might not be completely input when line 200 is executed, and line 200 could produce erroneous results.

Is it done yet? There are three ways to tell when a proceed mode I/O operation is complete. The first way was just discussed. When a variable that is involved in a proceed mode I/O operation is referenced, execution is automatically delayed. There's no need to check for the I/O completion before referencing a variable unless the execution delay will be a problem.

The second way is to use the ASK("IODONE",n) function. This function returns the following conditions:

1	if the specified logical unit number (n) is NOT busy performing an I/O operation.
0	if the specified logical unit number (n) is busy performing an I/O operation.
-1	if the specified logical unit number (n) is not open.

This function can be used at any point in a program to test whether an I/O operation is in progress. It can also be used to find out if a logical unit number has been OPENed.

The third way to detect the completion of an I/O operation is with the IODONE interrupt. A program can set up an IODONE condition and link it to a handler subprogram that is automatically called when an I/O operation is completed on a specified logical unit. The IODONE interrupt is discussed in more detail in Section 5.

Using the BUFFER Clause

The I/O buffer. When any I/O is performed, the 4041 initially receives data into or prints data from a special area of memory called an I/O buffer. On input, the 4041 accepts data into this buffer exactly as it is received. Then it translates the data from the received format in the I/O buffer to the internal format for storage in variables. On output, data is translated from the internal format to the format required for output and temporarily stored in the I/O buffer. Then, data is output to the device directly from the I/O buffer. Figure 4-14 illustrates the relationship of the variables, the I/O buffer, and the external device.

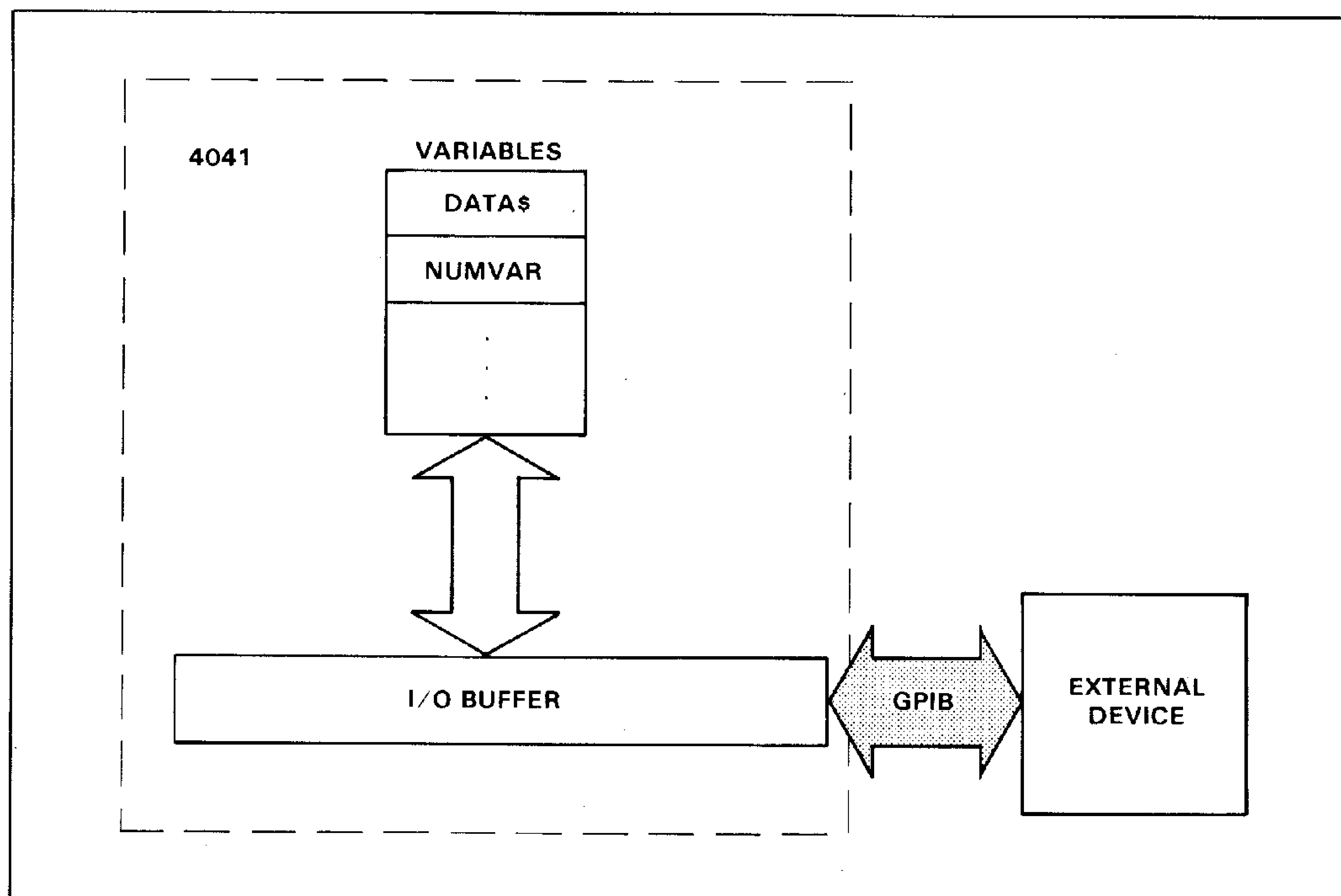


Fig. 4-14. *All I/O data passes through a special area of memory called the I/O buffer. On output, data is moved from the variables to the I/O buffer in preparation for output. On input, data is received into the I/O buffer. Then, the data is translated to internal storage format and moved into the appropriate variables.*

Defining an alternate I/O buffer. In a default I/O statement (without the BUFFER clause), the 4041 creates a temporary 514-byte I/O buffer in free memory. This buffer is transparent to the user. If the amount of data to be transferred exceeeds 514 bytes, the I/O is performed in 514-byte blocks. After each block, the 4041 processes the data in the I/O buffer before transferring more data. On input, the data in the I/O buffer is converted to internal storage format and stored in variables. On output, data is moved from the variables to the I/O buffer and formatted for output.

When the data in the I/O buffer is processed, another 514-byte block is transferred. This process is repeated until the data transfer is complete (Fig. 4-15).

The BUFFER clause for PRINT and INPUT allows you to specify a string variable to use as an I/O buffer. The string variable can be dimensioned to any length (up to 32,767 characters) to make the buffer as large as necessary. The data transfer speed can be increased by using a buffer large enough to hold the entire data transfer. This reduces the time required to transfer and process many smaller buffers. In addition, the contents of the buffer string are not deleted after the transfer is complete, so they are available for editing or other purposes.

PRINT and INPUT statements operate with the BUFFER clause the same as they do with the default I/O buffer. Variables are still filled as usual on INPUT and data is automatically translated from internal format to the correct format for output with PRINT. Figure 4-16 shows two programs that get an ASCII waveform from a 7D20 Programmable Digitizer. The program in part **a** uses the default I/O

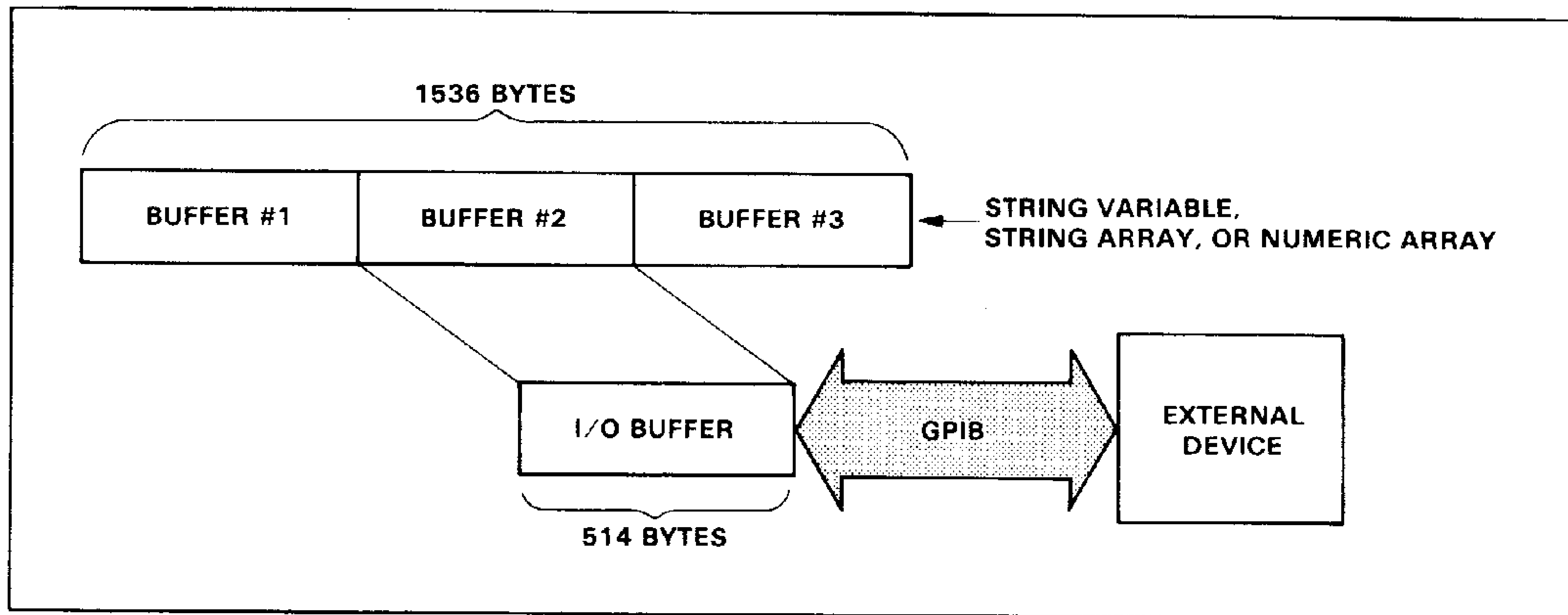


Fig. 4-15. *Data is input and output in blocks that are the length of the I/O buffer. After each block is transferred, the 4041 either empties the I/O buffer and moves the data to variables (INPUT) or refills the I/O buffer in preparation for more output (PRINT).*

buffer, while the program in part **b** uses the BUFFER clause. The programs perform the same function, but the program in part **b** will execute faster because the I/O is performed in a single block.

```
100  Dim waveform(1024)
110  Input prompt "CURVE?":waveform
```

(a) *Using the default buffer, I/O is performed in 514-byte blocks. After each block is transferred the 4041 empties the I/O buffer, storing the data in the specified variables.*

```
100  Dim waveform(1024),buffer$ to 5124
110  Input prompt "CURVE?" buffer buffer$:waveform
```

(b) *The BUFFER clause can be used to define a larger I/O buffer that allows the transfer to be completed in one block.*

Fig. 4-16. *The BUFFER clause allows you to define a larger I/O buffer. This makes a data transfer considerably faster, since the entire transfer can be completed in a single buffer.*

GETMEM and PUTMEM

String input and output is faster on the GPIB than most other forms of I/O because no translation or formatting is required—characters are stored exactly as they are received. But string input is a problem for numeric values. String variables cannot be used in numeric computations. Numeric values must be translated from the stream of ASCII digits received from the device to the internal numeric storage format (Fig. 4-17). This translation and formatting takes time and makes numeric input slower. As a result, when speed is critical it may be beneficial to receive data directly into a string variable.

Data stored in a string variable, whether through a BUFFER clause or through a regular INPUT statement, can be converted to individual string or numeric variables using the GETMEM statement. Conversely, data stored in numeric variables or string variables can be converted with PUTMEM to an ASCII string ready for output.

Using GETMEM. The GETMEM statement can be used at any point in a program, but the most common application is with proceed mode INPUT. In proceed mode, the INPUT statement can only accept data into a single string variable. After the INPUT has completed, GETMEM can take the data from this string, break it into separate variables, and translate the data into the format required for storage in numeric variables.

An example of using GETMEM is shown in Fig. 4-18. The 4041 is set to proceed mode and the INPUT statement in line 130 reads an entire

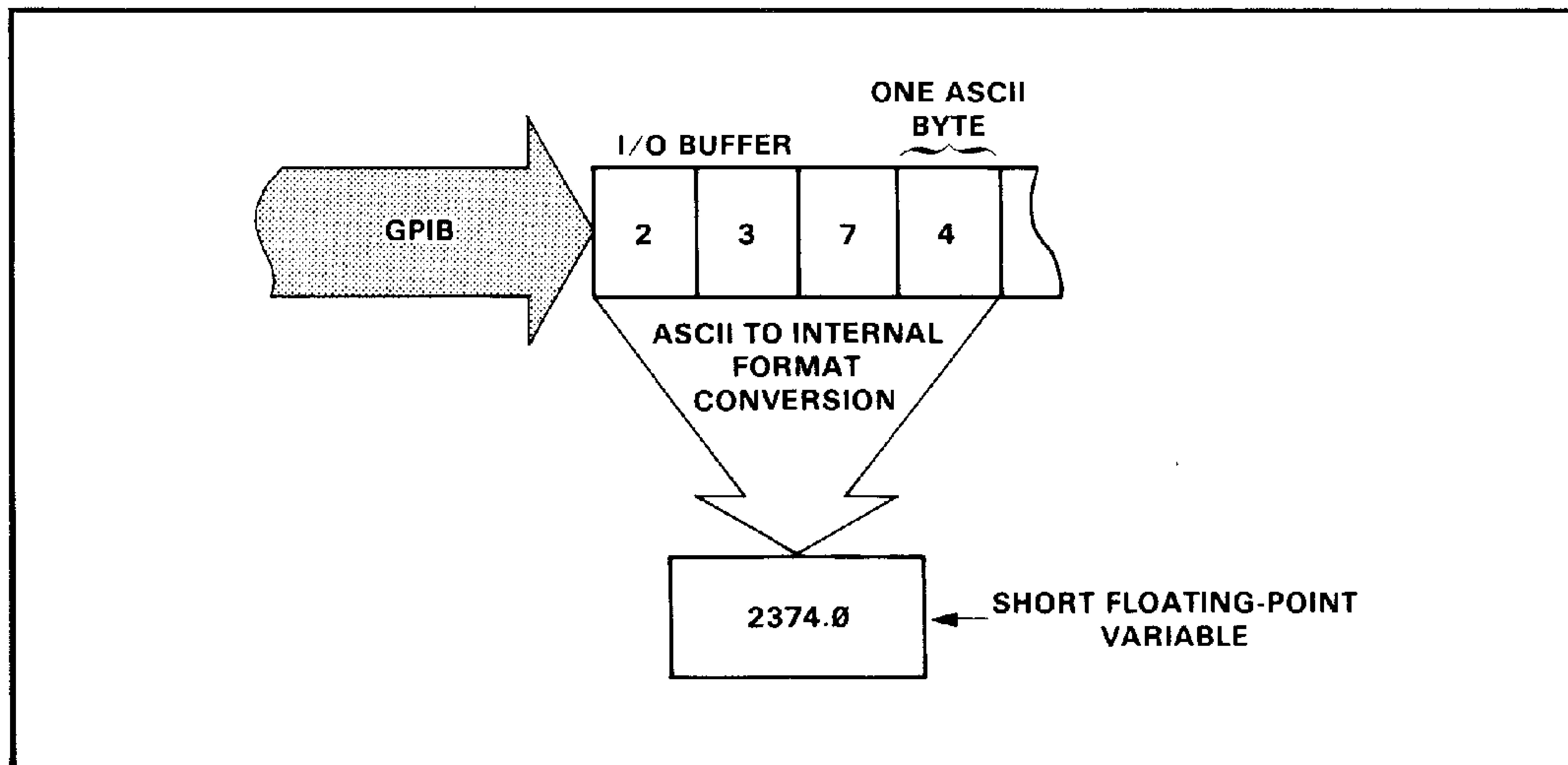


Fig. 4-17. *On input, numeric values are converted from the stream of ASCII digits to the internal numeric storage format (integer, short floating-point, or long floating-point). On output, the opposite process is performed.*

```
100    Set proceed 1
110    Dim buffer$ to 5124,waveform(1024)
120    Print #1:"CURVE?"
130    Input #1:buffer$
       :
200    Getmem buffer buffer$:waveform
```

Fig. 4-18. *The GETMEM statement can be used in proceed mode to translate the string data accepted with an INPUT statement and store it in the specified numeric and string variables.*

waveform into a single string variable. Each data point is delimited by a comma.

The input characters are received and stored directly in the string variable without any formatting or translation. While the input is in progress, the 4041 continues processing. When the I/O is complete, the GETMEM statement in line 200 converts the string into a series of numeric and string variables. Figure 4-19 shows a typical INPUT string and the resulting variables.

The GETMEM statement can use the DELS, DELN, and USING clauses just like an INPUT statement. DELS specifies an alternate delimiter for strings, DELN specifies an alternate delimiter for numeric data, and USING specifies the format of the data. These clauses allow you to control input in the same way as previously described for INPUT.

GETMEM with binary waveforms. One use of the USING clause with GETMEM is in translating binary block data read into a string variable with INPUT. In a data logging application, it is faster to read data into a string variable and store it directly on the mass storage device (tape or disk) without translating it to numeric format. After the data has been logged, it can be read back off the mass storage and translated to numeric format with GETMEM. Figure 4-20 shows a program that reads binary data from a 7612D Programmable Digitizer into a string variable and translates it to numeric format using GETMEM. The process of writing the data to tape and retrieving it is omitted for clarity.

Notice that the buffer string variable (BUFFER$) must be dimensioned to include the header characters, the two byte-count bytes, and the checksum, and the terminator character (if included). The INPUT statement requires a

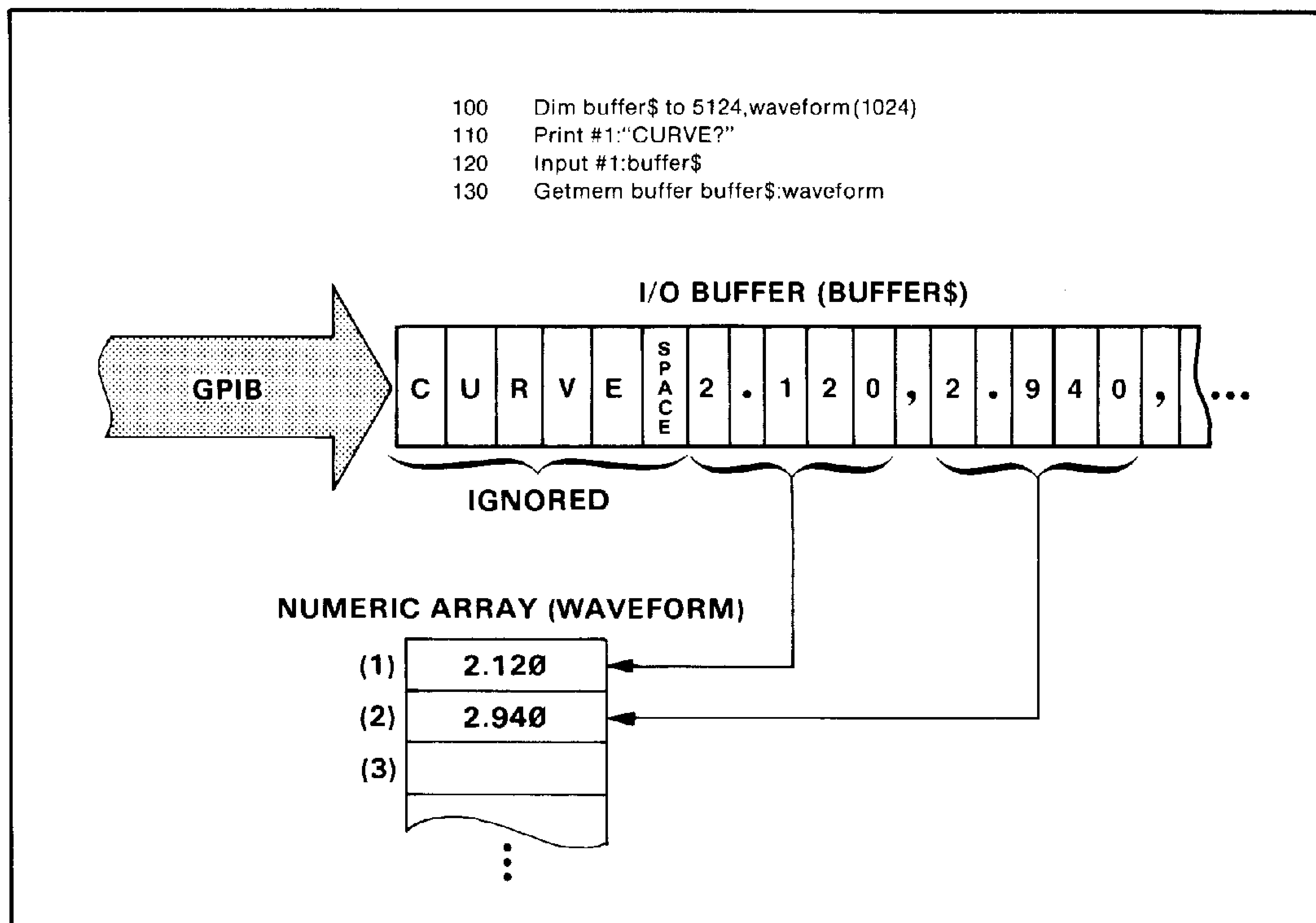


Fig. 4-19. *GETMEM translates data stored in a string variable the same way INPUT does when it accepts numeric or other data. In this case, GETMEM translates waveform data stored in a string variable (BUFFER$) to numeric array format.*

```
100 Open #3:"GPIB0(pri=3,sec=0,eom=<0>):"
110 Dim buffer$ to 2050,dummy$ to 1
120 Integer waveform(2048)
130 Input #3 prompt "READ A" buffer buffer$:dummy$
140 Getmem buffer buffer$ using "+8%":waveform
```

Fig. 4-20. *GETMEM can also be used to translate binary waveform data stored in a string variable to a numeric array.*

destination string for sytax consistency. In this case a dummy 1-character string (DUMMY$) is included in the list. After the input data is read into the I/O buffer, the first character is placed in DUMMY$.

The GETMEM statement in line 140 starts at the beginning of the buffer string and translates the data into numeric format, storing the results in the numeric array (WAVEFORM). Checksums and byte counts are automatically checked by GETMEM.

Using PUTMEM. PUTMEM does the opposite of GETMEM—it takes numeric or string variables, converts them to string format for output, and stores the converted characters in a specified string variable. The output of PUTMEM is in the same format that PRINT would produce with the same variable list. The EOH, EOA, and EOU characters are inserted into the output string just as they are in a PRINT statement.

Once a string buffer has been prepared for output with PUTMEM, the buffer can be sent directly with a PRINT statement. No translation is necessary since the buffer already contains data in the required

format. For example, the following statements prepare a binary block waveform for output using PUTMEM. The USING "+8%" clause generates a binary block with the byte count and checksum. The resulting block is stored in the buffer string ready for output.

```
100     Dim buffer$ to 1040,waveform(1024)
          :
200     Putmem buffer buffer$ using "+8%":waveform
```

Low-Level GPIB I/O

The 4041 BASIC PRINT and INPUT statements with all their clauses will perform most GPIB I/O operations. However in some special cases, more direct control of the GPIB may be helpful. The WBYTE (Write Byte), RBYTE (Read Byte), and SELECT statements in 4041 BASIC provide this direct "low-level" control of the bus.

The SELECT statement. WBYTE and RBYTE statements have default stream specs just like PRINT and INPUT do. The default stream spec for WBYTE, RBYTE, and POLL is set with the SELECT statement. When no SELECT statement has been executed, the stream spec is set for "GPIB0:". The stream spec can be set to the other GPIB port or other devices, such as the tape drive. A typical SELECT statement is shown below.

```
Select "GPIB1:"
```

ASK$("SELECT") returns the stream spec for the currently selected device.

The WBYTE statement. The WBYTE statement allows you to send any data byte on the GPIB or send any of the GPIB interface messages. The syntax of the WBYTE statement is shown below.

```
WBYTE #lun:wbyte-element,wbyte-element
```

where: LUN is the Logical unit number. If no LUN is specified, the currently SELECTed stream spec is used. The default stream spec for WBYTE is "GPIB0:"

WBYTE ELEMENT is either a numeric expression, a string expression, or a GPIB function.

WBYTE does not perform any automatic addressing or unaddressing like PRINT does. If device-dependent data is sent, the 4041 must be addressed to talk. When the 4041 is the controller-in-charge, the ATN(MTA) function can be used to talk-address the 4041. In talk/listen mode, the 4041 must be addressed to talk by the controller-in-charge.

Numeric elements in a WBYTE statement not only transfer data, but specify the state of the EOI bus line. If the numeric expression has a positive value, the EOI line is not asserted with that byte. If the byte is negative, EOI is asserted with the byte. When transferring a series of numeric values, the last one can be negated to indicate the end of the transmission by asserting EOI.

When a string data item is transferred using WBYTE, a numeric value immediately following the string expression in the WBYTE list can be used to control the state of the EOI line. If the value of the numeric expression is negative, EOI is asserted with the last character in the string. When the numeric value is positive or is omitted, EOI is not asserted. The numeric value is not transmitted—it simply controls the state of the EOI line.

WBYTE GPIB functions. Besides transferring device-dependent bytes from numeric variables or strings, WBYTE can send a variety of interface messages. To simplify sending these messages, a set of "GPIB functions" are provided. The following paragraphs briefly describe each GPIB function. The 4041 must be the controller-in-charge to execute any of these functions except EOI, SRQ, IFC, and REN. For IFC and REN, the 4041 must be the system controller (SC=YES), though it need not be the current controller-in-charge. If these conditions are not met, an error is generated.

Notice that the addressed GPIB functions (e.g., GET, GTL, SDC), do **not** automatically unaddress the listener when complete.

ATN(interface message[,interface message]...)
The ATN function allows you to send bytes with ATN asserted. The argument may be a numeric expression or one of the numeric GPIB functions (MTA, MLA, UNT, or UNL). Since the bytes are sent with ATN asserted, they are interpreted as interface messages. For example, WBYTE ATN(36) sends the listen address for a device set for primary address 4.

DCL
The DCL function sends the universal Device Clear message. WBYTE DCL is equivalent to WBYTE ATN(20).

EOI
The EOI function causes EOI to be asserted with the last byte of the preceding data element. The following example shows how EOI works.

```
WBYTE "SET?",EOI
```

The EOI function causes the 4041 to assert EOI when it sends the "?" character.

GET(listen address[,listen address]...)
The GET function sends the specified listen addresses followed by the Group Execute Trigger (GET) message. The listen addresses can be specified either as absolute listen addresses in the range of 32-62 or as primary addresses (0-30). Secondary addresses may also be included in the range 96-127.

GTL(listen address[,listen address]...)
The GTL function sends the specified listen addresses as discussed for the GET function. Then, the Go To Local (GTL) message is sent to the addressed listeners.

IFC(time in seconds)
The IFC function asserts the Interface Clear line on the GPIB for the number of seconds specified in the argument. The minimum time is 1E-4 (100 microseconds). If a value smaller than 1E-4 is specified, IFC is asserted for 100 microseconds.

LLO
The LLO function sends the universal Local LockOut message on the bus. This function is equivalent to sending ATN(17).

MLA
This function returns the value of the 4041's listen address (MA+32). It cannot be used by itself, since it only returns a value. Instead, MLA is used as an argument to the ATN function (e.g., ATN(MLA)). WBYTE ATN(MLA) can be used to address the 4041 to listen.

MTA
This function returns the value of the 4041's talk address (MA+64). It cannot be used by itself, since it only returns a value. Instead, MTA is used an an argument to the ATN function (e.g., ATN(MTA)). WBYTE ATN(MTA) can be used to address the 4041 to talk.

PPC(listen address,data-line,sense[,listen address,data-line,sense]...)
The PPC function configures the specified devices for a parallel poll. For each device configured, three arguments are specified: the listen address, the data line on which the instrument reports its status, and the sense value. The valid values for the listen address argument are discussed under the GET function. If a secondary address is required, four arguments are specified per instrument. The data line argument must evaluate to an integer between 0 and 8 which specifies the data bus line (DIO1-8) on which the instrument will report its status. If the argument evaluates to 0, a Parallel Poll Disable (PPD) message is generated.

The sense argument must evaluate to either a 1 or 0. If sense=1, the device will assert its assigned data line when polled if its Instrument STatus (IST) bit is set. Refer to the device manuals for information on the IST bit. The following example shows a typical PPC statement.

```
100     Wbyte ppc(5,6,0,24,7,1)
```

This statement configures the device at primary address 5 to assert data line 6 if its IST bit is false (SENSE=0). It also configures the device at primary address 24 to assert data line 7 if its IST bit is true (SENSE=1).

PPU
The PPU function sends the universal Parallel Poll Unconfigure message on the bus. This message clears a parallel poll configuration.

REN(state)
The REN function asserts or unasserts the REN bus line, depending on the value of the argument. If the absolute value of the argument is greater than or equal to 0.5, the Remote ENable (REN) line is asserted. If the absolute value of the argument is less than 0.5, REN is unasserted. The argument can be a constant or a numeric expression.

SDC(listen address[,listen address]...)
The SDC function sends a Selected Device Clear message to the devices whose listen addresses are specified in the arguments. Valid values for the address arguments are described under the GET function.

SPD
This function sends the Serial Poll Disable message to all devices on the bus. SPD unconfigures all devices for serial polls.

SPE
The SPE function sends the universal Serial Poll Enable message. This tells all the devices on the bus to prepare to respond with a status byte when they are addressed to talk.

SRQ(status byte)
The SRQ function causes the 4041 to assert or release SRQ. The argument is the status byte that the 4041 will send when it is serial polled by the controller-in-charge. To assert SRQ, the value of the argument must evaluate to an integer between 64 and 127 or between 192 and 255. If the value is outside this range, an error is generated. To release SRQ, the argument must evaluate to zero.

The 4041 must NOT be the controller-in-charge when this function is executed. Otherwise, an error is generated.

TCT(talk address)
The TCT function sends the specified talk address followed by the addressed Take ConTrol interface message. This message tells the addressed device to take over the role of controller-in-charge. After this function is executed, the 4041 is no longer the controller-in-charge. As a result, this must be the last function executed that requires asserting ATN.

UNL
This function returns the value of the UNListen universal message (63). UNL cannot be used by itself, since it only returns a value. Instead, the function is used as an argument to the ATN function (e.g., ATN(UNL)). WBYTE ATN(UNL) can be used to unaddress all listeners on the bus.

UNT
This function returns the value of the UNTalk universal message (95). UNT cannot be used by itself, since it only returns a value. Instead, the function is used as an argument to the ATN function (e.g., ATN(UNT)). WBYTE ATN(UNT) can be used to unaddress the current talker.

Transfers among GPIB instruments. One common use of the low-level GPIB functions is for setting up data transfers between two devices on the bus without involving the 4041. To perform such a transfer, one device must be addressed to talk. This device might be a waveform digitizer or other measurement instrument. Another device must be addressed to listen to receive the data sent from the instrument. The listener might be a tape drive used to log data.The example program shown in Fig. 4-21 illustrates how this is accomplished.

```
100   Open #1:"GPIB0:"
110   On eoi(1) then gosub 190
120   Enable eoi(1)
130   Digpri=4
140   Tappri=10
150   Listen=32
160   Talk=64
170   Wbyte atn(digpri+talk,tappri+listen)
180   Wait
190   Wbyte atn(unt,unl)
200   Branch 210
210   .
      :
```

Fig. 4-21. *This program sets up a data transfer between two GPIB devices without involving the 4041 in the transfer. The 4041 addresses one device to talk and others to listen. Then it waits for the talker to assert EOI at the end of the message.*

First, the EOI interrupt condition is linked to a routine starting at line 200 and the interrupt is enabled. The occurrence of EOI signals the end of the data transmission from the talker and causes the 4041 to branch to line 200. The EOI interrupt is described in more detail in the next section.

Next, the primary addresses and the talk and listen offsets are set up in variables. The talk address of a device is simply its primary address plus the TALK offset. The listen address of a device is its primary address plus the LISTEN offset.

The WBYTE statement in line 160 sends the digitzer's talk address and the tape drive's listen address. The digitizer should have already been told to send data, and the tape drive should have already been told to accept binary data and store it in a file.

At this point, the talker takes over the bus and begins sending it's data to the tape drive. The 4041 is not involved in the transfer, so the WAIT statement in line 180 delays further execution. When the digitizer finishes sending data, it asserts EOI. The interrupt condition causes program execution to branch past

the WAIT statement to line 190. Line 190 sends the universal UNT and UNL messages to unaddress the digitizer and tape drive. Execution proceeds at line 210.

The RBYTE statement. RBYTE allows you to receive bytes directly from the GPIB. RBYTE does not perform any of the automatic addressing or unaddressing that INPUT does. If the 4041 is the controller-in-charge, it must address itself to listen with WBYTE ATN(MLA) before beginning data transfers. If the 4041 is not the controller-in-charge, the controller must address it to listen. If the 4041 is not addressed to listen when an RBYTE statement is executed, an error is generated.

The data received from the bus is stored directly in the numeric or string variables as it is received from the bus. This can be useful for transferring data formats that are not supported with INPUT or INPUT USING.

Bytes stored in numeric variables are assumed to be integers. The value of the numeric variable also indicates the state of EOI. If EOI is asserted with the byte, the value stored in the variable is negative. If EOI is not asserted, the value is positive.

When numeric arrays are read with RBYTE, one byte is read per array element. For string variables, one byte is read per character in the string variable. The default length for strings read with RBYTE is one character. Longer strings can be DIMensioned. String input is terminated when either the string is filled or when EOI is asserted.

If a numeric variable is specified following a string variable in RBYTE, the state of EOI is stored in the numeric variable. No data is read for this variable. If EOI was asserted with the last byte of the string, the variable is set to a value of 1. If EOI is not asserted, the variable is set to a value of 0.

The example in Fig. 4-22 shows a program that sends a SET? query to an FG 5010 Programmable Function Generator and gets the reponse using WBYTE and RBYTE.

```
100   Fg5010=24
110   Listen=32
120   Talk=64
130   Integer Eoistate
140   Dim setting$ to 200
150   Wbyte atn(mta,fg5010+listen)
160   Wbyte "SET?",eoi
170   Wbyte atn(unt,unl,mla,fg5010+talk)
180   Rbyte setting$,eoistate
190   Wbyte atn(unt,unl)
```

Fig. 4-22. *This program sends a SET? query to an FG 5010 Programmable Function Generator and gets the response. The same operation could be performed with an INPUT PROMPT statement, but for the sake of example, this program uses only RBYTE and WBYTE.*

Lines 100-170 set up the transfer, addressing the FG 5010 and transferring the SET? query. Then, line 180 reads the response string. The state of the EOI line when the last byte was transferred is stored in EOISTATE. This is a convenient way to tell whether the string input terminated as the result of filling the string variable or because the talker finished transmitting data. Line 180 unaddresses the FG 5010 and the 4041.

The 4041 as a Talker/Listener

Remember from Section 2 that there are two kinds of controllers—the system controller and the controller-in-charge. The system controller is the default controller-in-charge. However, control of the system can be passed to any other device that is capable of assuming control. After control is passed, the new device becomes the controller-in-charge. Any number of devices in the system may be capable of controlling the system. However, there can be only one system controller and only one device can be the controller-in-charge at any time (Fig. 4-23).

The 4041 is capable of acting as the system controller and the controller-in-charge on either or both of its GPIB ports (only Option 1 units have a second GPIB port). It powers up with the SC (System Controller) GPIB parameter set to YES on both ports. In other words, it defaults to being the system controller. However, the 4041 can be told NOT to assume the role of system controller by setting the SC parameter to SC=NO.

When the 4041 is not the controller-in-charge, it can receive control of the bus from the controller-in-charge. The controller-in-charge sends the 4041's

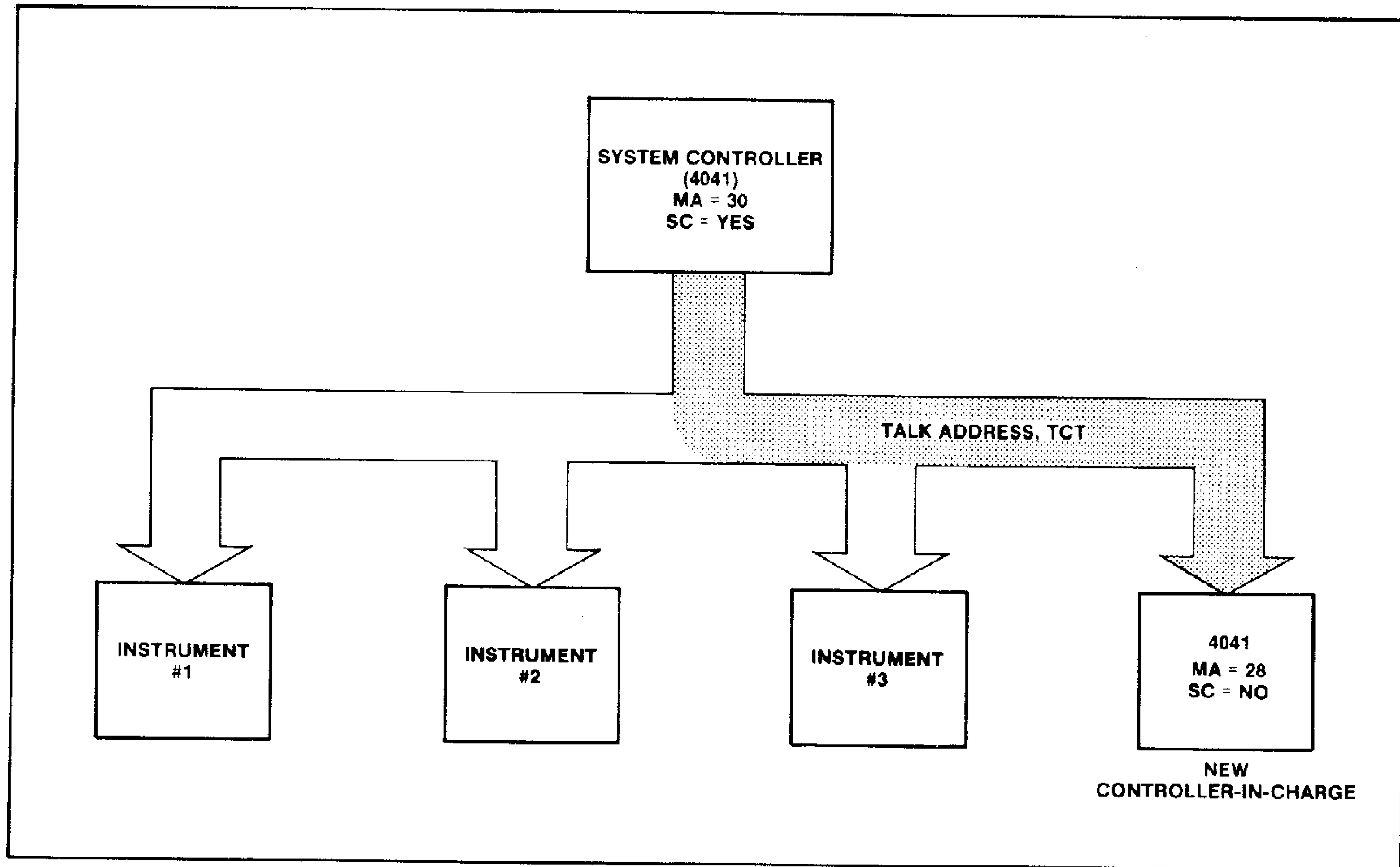


Fig. 4-23. *The system controller is the default controller-in-charge. However, any device on the bus that is capable of assuming control can become the controller-in-charge.*

talk address followed by the TCT (Take Control) interface message. Then the 4041 becomes the controller-in-charge. The 4041 can pass control to another device using the same protocol.

The system controller can regain control if the controller-in-charge passes control to it using the TCT message, or the system controller can regain control at any time by asserting IFC. Only the system controller can assert IFC.

The two GPIB ports on a 4041 equipped with Option 1 are independent. The 4041 can be the system controller on one and a talker/listener on the other. Each port can have a separate primary address and can pass and receive control separately.

Talker/Listener programming. When the 4041 is not the controller-in-charge (whether or not it is the system controller), it can act as a normal talker or listener on the bus. It can be addressed to talk or listen just like any other device. The MA (My Address) GPIB parameter sets the 4041's primary bus address.

PRINT and INPUT statements work the same in talk/listen mode as when the 4041 is the controller-in-charge with one important difference. In talk/listen mode, the 4041 does not control the bus and cannot send talk or listen addresses. It can only send data and receive data as instructed by the controller-in-charge. As a result, input and output must be directed to the appropriate GPIB port without primary or secondary addresses. This can be accomplished by OPENing a LUN with primary address 31 (the default). Primary address 31 tells the 4041 to direct I/O to the assigned port but not to generate any addressing sequence. Figure 4-24 shows how this is implemented in 4041 BASIC.

The OPEN statement in line 100 opens a LUN with the default primary address (31) and a carriage return as the EOM character. Later, when the controller addresses the 4041 to talk, line 200 sends

```
100    Open #1:"GPIB0(eom=<13>):"
          :
200    Print #1:"DATA TO SEND ON THE GPIB"
          :
300    Input #1:data$
```

Fig. 4-24. *The PRINT and INPUT statements function the same way in talk/listen mode as they do when the 4041 is the controller-in-charge, except that they cannot generate any addressing or unaddressing sequence.*

the message "DATA TO SEND ON THE GPIB." This PRINT statement generates no addresses. Data is transferred from the 4041 to the listeners addressed by the controller.

When the 4041 is addressed to listen, the INPUT statement in line 300 receives a string of data from the talker and stores it in the string variable DATA$. Again, the 4041 generates no addresses.

Knowing when to talk and when to listen. When the 4041 is in Talker/Listener mode, there are at least two programs executing in the system—one in the controller-in-charge and one in the 4041. The key to making the system work is coordinating these two programs. The 4041 must be prepared to talk (PRINT or WBYTE) when the controller addresses it to talk, and it must be prepared to listen (INPUT or RBYTE) when the controller addresses it to listen.

Since the 4041 is not in control of the bus, it can't initiate data transfers. It must wait for the controller to address it to talk or listen. How, then, does the 4041 know when to execute a PRINT or INPUT statement, now knowing when it will be addressed?

There are three ways to handle this dilemma. First, the TIM (TIMeout) parameter can be set to a very large value (the default for open LUNs is 2.14748 E+7 seconds—about 250 days!). When the 4041 executes a PRINT statement, it simply waits to be addressed to talk. If the controller does not address the 4041 before the timeout period is over, the 4041 generates a timeout error. With a long timeout, that isn't likely.

The same technique can be used on INPUT. The 4041 waits to be listen addressed before beginning to accept data from the bus. This works as long as the controller doesn't address the 4041 to listen when it's waiting to talk or vice versa.

A more reliable technique for finding out if the 4041 has been addressed is to use the ASK$("LU",n) function to get information on the current state of an open LUN. One of the parameters returned from this request is the TL (Talk/Listen) value. The TL value is a number between 0 and 62. It is a bit-encoded number that indicates several bus status conditions. Table 4-3 shows the values for the TL parameter and their meanings. More than one of the conditions can be true, in which case the TL value is the sum of the individual bit values.

TABLE 4-3
TL PARAMETER VALUES

Value	Meaning
0	Bus unconfigured
1	Serial poll active; SPE has been sent (only valid if the 4041 is the CIC*).
2	4041 is talk addressed.
4	4041 is listen addressed.
8	Some device is talk addressed (may be the 4041; only valid if the 4041 is the CIC).
16	At least one device is listen addressed (may be the 4041; only valid if the 4041 is the CIC).
32	Parallel polls is active (only valid if the 4041 is the CIC).

**CIC = Controller-in charge.*

A program can read the value of the TL parameter and test bits 2 and 3 of the TL parameter to see if the 4041 is addressed to talk or listen and then execute the appropriate PRINT or INPUT statements. Figure 4-25 shows how the bits are tested with a BAND (Binary AND) operator.

First, the program opens LUN 1 with the default primary address. A string variable (GPIBSTA$) is dimensioned to hold the GPIB status information returned from the ASK$ function. Line 120 gets the status information and line 130 extracts the value of the TL parameter from the status string. Then, line 140 tests bit 2 (4041 talk addressed). If bit 2 is set,

```
100     Open #1:"GPIB(eom=<10>):"
110     Dim gpibsta$ to len(ask$("lu",1))
120     Gpibsta$=ask$("lu",1)
130     Tl=VALC(GPIBSTA$,POS(GPIBSTA$,"TL=",1))
140     If tl band 2 then gosub talk
150     If tl band 4 then gosub listen
160     Goto 120
170 Talk:   rem ** PRINT STATEMENTS GO HERE **
180     Return
190 Listen:  rem ** INPUT STATEMENTS GO HERE **
200     Return
```

Fig. 4-25. *This simple program continually checks the status of the GPIB port using the ASK$("LU",1) function. If the 4041 is addressed to talk or listen, the appropriate subroutines are called to PRINT or INPUT data.*

the IF statement in line 140 sends control to the TALK subroutine. This routine could contain PRINT statements to send data on the bus. If bit 4 is set, the 4041 is addressed to listen, and line 150 calls the LISTEN subroutine. When the 4041 is neither talk addressed nor listen addressed, it simply loops, waiting to be addressed.

Both of the previous techniques require some wasted time either waiting to be addressed or continually testing the interface status. A more efficient technique involves using two special interrupt conditions provided in 4041 BASIC. In this discussion, these special interrupts are described in general and examples of their use are provided. (Refer to Section 5 for more detailed information.)

4041 BASIC includes MTA (My Talk Address) and MLA (My Listen Address) interrupt conditions. A program can set up a handler subroutine or subprogram that is automatically called when the specified condition occurs. The MTA condition occurs when the 4041 receives its talk address. The MLA condition occurs when the 4041 receives its listen address. Figure 4-26 shows the program from Fig. 4-25 converted to use the MTA and MLA interrupts instead of the ASK$ function.

The ON statements in line 110 and 120 set up the interrupt conditions. They set up a link between the interrupt condition and the specified subroutine or subprogram. This subroutine or subprogram is called a "handler" because it handles the interrupt. Line 130 enables the MTA and MLA interrupts.

```
100     Open #1:"GPIB0(eom=<10>):"
110     On mta(1) then gosub talk
120     On mla(1) then gosub listen
130     Enable mta(1),mla(1)
          :
500 Talk:   rem ** PRINT STATEMENTS GO HERE **
          :
590     Resume
600 Listen:   rem ** INPUT STATEMENTS GO HERE **
          :
690     Resume
```

Fig. 4-26. *The MTA and MLA interrupt conditions provide a simple, efficient way of detecting when the 4041 is addressed to talk and when it is addressed to listen.*

The handler is not called when the ON statement is executed; the statement just sets up a link to the handler. Execution continues normally until the interrupt occurs. Then, program execution is temporarily suspended and the handler is called. When the handler finishes its work, execution resumes at the point it was interrupted.

Storing and Retrieving Data on Tape

The 4041's internal DC-100 tape drive can store data and programs. This discussion only considers the use of the tape drive for storing data—waveforms or other measurement data read from instruments. Complete information on using the tape is available in the 4041 Programmer's Reference Manual.

Unlike most tape systems, the 4041's tape is a directory-structured device. It has a directory at the beginning of the tape that lists the contents of the tape by file name. The directory also includes information about the position of the file on the tape, the length of the file, the file type, and the time and date of the last revision of the file. As a result, the position of a file on the tape is transparent to the user. You can access a file by name without knowing where it is on the tape.

The 4041 tape drive can be operated in two basic I/O modes: logical I/O mode and physical I/O mode. In logical mode, all I/O with the tape drive is performed within the confines of a file. A file is

opened with the OPEN statement and segments of information called "records" are read from and written to the file with standard PRINT and INPUT statements. The physical location of the file on the tape is transparent to the user. All you have to do is OPEN the file by name.

In physical I/O mode, information is read from or written to specific physical locations on the tape using RBYTE and WBYTE statements. No file structure is used and the user is responsible for knowing the location of the data on the tape. The I/O mode is set with the PHYsical parameter in the OPEN statement. If PHY=YES is specified, the tape is set to physical mode. The default is PHY=NO (logical mode).

In addition to these two I/O modes, the 4041 can store data on tape in two formats. Information can be stored in ASCII or in the internal 4041 data format, called ITEM format. The data format is selected with the logical FORMAT parameter in the OPEN statement. The default is ASCII.

Reading and writing ASCII data. Consider first the simplest case—reading and writing ASCII data in logical mode. The first step is to OPEN the file. A typical OPEN statement is shown below.

```
OPEN #32:"filnam(open=new)"
```

The OPEN statement associates a file with a logical unit number and specifies logical and file parameters. If the OPEN=NEW parameter is specified, a new file is created. The default OPEN parameter is OPEN=OLD, where an existing file is opened. This parameter may also be set to OPEN=REPlace to delete the existing file and replace it with the new one or OPEN=UPDate to add to the end of an existing file.

A variety of other logical parameters can be specified in the OPEN statement to set the size and format of the file, the terminator characters (EOH, EOU, EOA, EOM), and other parameters. The complete parameter list is provided in the 4041 Programmer's Reference Manual.

Once the file is OPEN, data can be written to the tape using the PRINT statement and read from the tape using the INPUT statement. Data transfers follow the same rules as for I/O with other devices using PRINT and INPUT. The only difference is in the terminator characters. The EOA and EOH characters default to zero (no output), the EOU character defaults to space, and the EOM defaults to carriage return. Numeric input can be delimited by space, tab, comma, colon, semicolon, or the EOM character. For string input, the only valid terminator is the EOM character.

Figure 4-27 illustrates a simple data transfer with the tape. A file is opened called "WFMDAT" and ASCII waveform data is written to the tape. Then the file is closed, re-opened, and the data read back in.

```
100    Dim waveform(1024)
110    Open #32:"WFMDAT(open=new)"
120    Input #3 prompt "CURVE?":waveform
130    Print #32:waveform
140    Close all
         .
         .
         .
300    Open #32:"WFMDAT"
310    Input #32:waveform
```

Fig. 4-27. *Writing and retrieving ASCII data is easy—just OPEN the file and PRINT to it or INPUT from it.*

A special case—binary waveforms in string variables. Figure 4-28 shows another example of writing waveform data. In this case, a binary waveform is input into a string buffer and the string is written directly to tape. The data is transferred and written to tape faster than the previous example, because no translation is required—the data is written directly from the string buffer. Since the binary waveform may contain a byte that is equivalent to the carriage return code, the EOM character must be set to null (0). This means that data accepted from the GPIB is terminated only by the EOI signal. When the data is written to the tape, no terminator is appended.

So far, so good. But what about reading the data back from the tape? Since there is no terminator at the end of the string, the string variable that receives the data from the tape must be dimensioned to the **exact** length of the string. The length of the string must be known ahead of time.

Detecting the end of the file. In some cases, you may not know exactly how many records a file

```
110    Open #32:"WFMDAT(open=new,clip=yes,eom=<0>)"
120    Dim buffer$ to 2060
130    Input #3 prompt "ARM A;READ A:" buffer buffer$:buffer$
140    Print #32:buffer$
150    Bufsiz=len(buffer$)
160    Close all
       :
300    Open #32:"WFMDAT(eom=<0>)"
310    Dim buffer$ to bufsiz
320    Integer waveform(1024)
330    Input #32 buffer buffer$ using "+8%":waveform
340    Close all
```

Fig. 4-28. *Binary waveform data can be input to strings and written to tape directly. However, the EOM character must be set to null (0) and you must know exactly how long the string is when you read it back since there is no terminator.*

contains. As a result, a program might be written to read data from the tape until the end of the file is reached. The EOF function provides a means of testing for the end of file. This function returns a value of 0 if the file associated with the specified LUN is not at the end of the file, or a value of 1 if it is at the end of the file. The following program segment illustrates the use of the EOF function.

```
100    Open #32:"DATFIL"
110    Loop: Input #32:data$
       :
200    If eof(32)=0 then goto loop
```

The program continually reads data from the tape file until the end of the file is encountered. Then, the EOF function in line 200 returns a value of 1 and execution continues with the next statement after line 200.

The TYPE function. The TYPE function returns an integer from 0 through 4 indicating the type of data stored as the next item in the file. Table 4-4 shows the possible values returned by the TYPE function.

The TYPE function can be used in ITEM format files to determine the type of data stored in the next unit of a file. This allows you to read data from a file into the proper string and numeric variables without knowing the format of the file ahead of time. Figure 4-29 shows how the TYPE function might be used to branch to different routines based on the type of data encountered in a file.

TABLE 4-4
VALUES RETURNED BY THE TYPE FUNCTION

Value	Meaning
0	Empty file or file not open
1	End-of-file character
2	ASCII numeric data, string data, or program
3	ITEM format short floating-point numeric data
4	ITEM format string data
5	ITEM format integer numeric data
6	ITEM format long floating-point numeric data
7	ITEM format program

ITEM format files. Data written in ITEM format is written directly in the internal 4041 memory storage format. It is not translated to ASCII characters as in ASCII format files.

For the most part, the difference between ASCII and ITEM files is transparent to the user. ITEM format files may be a little smaller than the equivalent ASCII format file, and ITEM format programs load faster than ASCII format programs because they are already translated into internal 4041 storage format.

One other difference between ITEM and ASCII files is that in ITEM format files, the TYPE function can differentiate between the various types of numeric data and string data. In ASCII files, all data is written as a string of ASCII characters. As a result, the TYPE function can't tell the difference between numeric data and string data. This distinction in ITEM format files can be useful for reading data from a file when you don't know its format.

The choice of which format to use depends on the application. Though ITEM files are smaller and may load faster, they can't be COPYed to a terminal screen or other ASCII devices directly, since they do not contain ASCII characters.

Physical mode I/O. When data is written in physical mode, the user has complete control (and responsibility) for the location of the data. No file structure is maintained and the data cannot be accessed by a file name (unless the file name is manually inserted into the directory in the correct format). Essentially, the tape appears to be a long string of 256-byte records in physical mode. An intimate knowledge of the data format is essential when performing any physical mode I/O with the tape.

```
100   Open #33:"DATFIL"
110   Dattype=typ(33)
120   Goto dattype of 200,300,400,500,600,700,800
130   Rem ** Empty file routine **
      :
200   Rem ** End-of-file routine **
      :
300   Rem ** ASCII character routine **
      :
400   Rem ** ITEM short floating-point numeric data routine **
      :
500   Rem ** ITEM string data routine **
      :
600   Rem ** ITEM integer numeric data routine **
      :
700   Rem ** ITEM long floating-point numeric data routine **
      :
800   Rem ** ITEM format program routine **
```

Fig. 4-29. *The TYPE function can be used to determine the type of data stored in a file. In this example, the value returned by the TYPE function is used as an index for a computed GOTO statement that calls an appropriate routine for each data type.*

CAUTION

There is no protection in physical I/O mode against corrupting the data structure of the tape. Any record can be read or written. As a result, indiscriminate use of physical I/O can result in a tape that cannot be read in logical mode.

Physical I/O mode is useful for special tape operations (e.g., copying tapes, rearranging files to collect contiguous free space, etc.) and it can be used in high-speed data-logging applications. However, for most normal applications, logical I/O is much simpler and more practical to use.

PRINT and INPUT statements cannot be used for physical mode I/O. Physical I/O is performed with WBYTE and RBYTE statements. The format of the WBYTE and RBYTE statements for physical tape I/O are:

```
Wbyte #lun:recnum,string$
Rbyte #lun:recnum,string$
```

The specified LUN must have previously been assigned to the TAPE with an OPEN statement. However, **a file name must not be specified**. No files should be open during physical tape I/O. The recnum is the physical record number on the tape where the data is to be read from or written to. Finally, the string variable (string$) is the data to be written (WBYTE) or the variable where data read from the tape will be stored (RBYTE). Only string data may be written or read with WBYTE and RBYTE.

Since RBYTE and WBYTE transfer complete records, the string data is assumed to be 256-bytes long. If the string in a WBYTE statement is longer than 256 characters, only the first 256 characters are written. If the string is shorter than 256 characters, the remainder of the record is filled with zeros. RBYTE always reads 256 bytes. If the string is dimensioned to less than 256 characters, the characters that don't fit in the string are lost.

RBYTE and WBYTE ignore all file boundaries.

Section 5—Processing Interrupts in 4041 BASIC

Sometimes conditions occur that may require special attention from the 4041. These conditions are called "interrupts" and they may occur for a variety of reasons. For example, a GPIB device may assert the SRQ (Service Request) line, indicating that it needs special service from the controller, or a 4041 front-panel user-definable key might be pressed.

4041 BASIC has a complete set of statements designed to handle these interrupt conditions. This section introduces you to the concept of interrupts and describes the statements provided in 4041 BASIC for interrupt handling. The GPIB interrupt conditions are described in detail. Examples of processing GPIB interrupts are also provided. Interrupts from non-GPIB sources are briefly described. More information on the other interrupt conditions can be found in the 4041 Programmer's Reference Manual.

Interrupt Conditions

Interrupts can be caused by one or more of the following conditions:

- One or more of the following conditions occurs on the GPIB:
  - the SRQ line is asserted
  - the EOI line is asserted
  - the IFC line is asserted
  - receipt of the DCL message
  - receipt of the 4041's MLA
  - receipt of the 4041's MTA
  - receipt of the TCT message
- Pressing the ABORT key on the front panel or program development keyboard or receiving a control-C from the COMM port.
- An error condition that occurs while executing a program or immediate mode statements.
- The completion of an I/O operation.
- Pressing a front-panel user-definable key.

All of these interrupt conditions can be detected within a program. When an interrupt is detected, a special routine, called a handler, is automatically called.

What is a Handler?

A handler is a special subroutine or subprogram that is called when the interrupt condition specified in an ON...THEN CALL or ON...THEN GOSUB statement occurs. Handlers are just like any other subroutine or subprogram with two exceptions.

First, handlers are not called with a regular CALL or GOSUB statement and they are not terminated with a RETURN statement. Instead, the handler is called with an ON statement and is terminated with one of the special handler exit statements. These statements are described later.

The second difference is that a handler subprogram must not require that any parameters be passed to it through the CALL statement. Global variables may be used to pass values and the SUB statement that begins a subprogram handler may define local variables.

Calling a Handler

All interrupts, regardless of their source, are asynchronous. In other words, you don't know when they will occur during the execution of a program. As a result, there must be some facility for detecting the occurrence of an interrupt and calling the appropriate handler. The ON statement in 4041 BASIC allows you to specify the name or line number of the handler that should be called when a specified interrupt condition occurs. The syntax of the ON statement is:

ON interrrupt condition **THEN** { **GOSUB** line number / **CALL** subprogram name }

When the ON statement is executed, nothing obvious happens, but the 4041 sets up an internal link between the specified interrupt condition and the handler subroutine or subprogram. When the interrupt occurs, the 4041 temporarily suspends execution of the current program segment and begins executing the handler. At the end of the handler routine, control can be returned using several handler exit statement options. These options are described later in this section.

A Logical Unit Number (LUN) can also be specified in an ON statement for a GPIB condition or the IODONE condition. If a LUN is specified with a GPIB condition, the handler is called when the specified condition occurs on the LUN or the port associated with that LUN. For example, if a program contains the statements:

```
100    Open #10:"GPIBO(pri=10):"
110    On srq(10) then call hansrq
120    Enable srq(10)
```

If any device on the GPIB0 bus asserts SRQ, the handler routine (HANSRQ) is called. Notice that **any** device on GPIB0 could have asserted SRQ, but the 4041 calls HANSRQ regardless of which device asserted SRQ. If the LUN is not specified as in the first example, the currently SELECTed GPIB stream spec is used. On power-up, the standard GPIB port is the SELECTed default device for GPIB operations. The SELECT statement can be used to change this default assignment (see Section 4 for more information on SELECT).

When a LUN is specified with an IODONE interrupt condition, the specified handler is called when an I/O operation is completed on the specified LUN. If a LUN is not specified, the IODONE condition applies to the console device. The handler is called when an I/O operation completes on the console device.

System Handlers

Some interrupt conditions are already linked to a system handler routine that is invisible to the user. For example, the system error handler reports a message when an error occurs. Alternate user-written handlers can be linked to the condition using the ON statement. If an ON statement is not executed, the default system handler is called when the condition occurs. Other interrupt conditions do not have a pre-defined handler. Table 5-1 lists the interrupt conditions and shows which ones have pre-defined handlers and which ones do not.

Enabling an Interrupt

Most interrupt conditions are disabled by default at power-up. If you want to recognize an interrupt condition, you must explicitly enable it (unless the particular condition is enabled by default at power-up). The ENABLE statement accomplishes this. The syntax of the ENABLE statement is:

```
ENABLE interrupt condition[,interrupt condition,...]
```

Disabling an Interrupt

You can also disable most interrupt conditions if they were enabled at power-up or explicitly enabled with an ENABLE statement. The DISABLE statement is just like the ENABLE statement, except that it disables the specified interrupt condition. The syntax of DISABLE is the same as for ENABLE. Some interrupt conditions, such as Error, cannot be disabled.

Table 5-1 shows which interrupt conditions are linked to a system handler at power-up, which ones are enabled at power-up, and which ones can be disabled.

Exiting from a Handler

When the execution of a handler subroutine or subprogram is complete, several options are available for transferring control back to the interrupted program segment or to another point. Some of the options can only be used with handlers for certain interrupt conditions. For example, the RETRY statement can only be used to exit from error handlers. Table 5-2 summarizes the exit statement options and shows which ones can be used for each interrupt condition.

TABLE 5-1
DEFAULT ENABLED INTERRUPT CONDITIONS

Interrupt Condition	Linked at power-up?	Enabled at power-up?	Can be disabled?
ABORT	Yes	Yes	Yes
ERROR	Yes	Yes	No
GPIB CONDITIONS	No	No	Yes
IODONE	No	No*	No
KEYS	No	No	Yes

* The IODONE condition is automatically enabled when the 4041 is set to proceed mode (SET PROCEED 1) and it is automatically disabled when proceed mode is disabled (SET PROCEED 0).

TABLE 5-2
HANDLER EXIT STATEMENT OPTIONS

Interrupt Condition	Exit Statement				
	Advance	**Branch**	**Monitor**	**Resume**	**Retry**
ABORT		X	X		
ERROR	X	X	X		X
GPIB CONDITIONS		X		X	
IODONE		X		X	
KEYS		X		X	

The ADVANCE Statement. The ADVANCE statement can only be used to exit from error handlers. It returns control to the statement after the one that caused the error.

The BRANCH Statement. The BRANCH statement can be used to exit from a handler for any of the interrupt conditions. BRANCH passes control to a specified line number. This line number can be any line within the active call sequence (see Appendix C for a definition of the active call sequence).

The MONITOR Statement. The MONITOR statement returns control to the system abort or error handler. MONITOR can only be used to exit from abort or error handlers. Since it returns control to the system handlers, the subroutine can "intercept" errors or abort requests. For example, a handler could intercept and process certain errors while reporting others normally. The MONITOR statement allows you to pass control to the system handlers for abort processing or error reporting.

The RESUME Statement. The RESUME statement can be used to exit from a GPIB handler, a user-definable key handler, or an IODONE handler. RESUME returns program control to the point where execution was interrupted.

The RETRY Statement. The RETRY statement can only be used to exit from error handlers. It retries the statement that caused the error and continues processing from that point.

Nested Handlers

In some situations, you may want to use different handlers to process the same interrupt at different points in a program. You can include more than one ON statement for an interrupt condition in your program. Each time a new ON statement is executed, any previous ON statements for that condition are superseded. Figure 5-1 shows an example program that uses several ON statements at various points in a single program segment.

```
100   On srq then gosub handler1
        :
150   Rem ** handler1 in effect **
        :
200   On srq then call handler2
        :
250   Rem ** handler2 in effect **
        :
300   On srq then call handler3
        :
350   Rem ** handler3 in effect **
        :
```

Fig. 5-1. *Executing an ON statement supercedes any previous links for the same interrupt condition.*

When an ON...CALL statement is included in a subprogram or user-defined function, the link is in effect for the subprogram or function and for any subprograms or user-defined functions it calls. However, when the subprogram returns to its caller, the handlers revert to the definitions specified by the caller (except for ABORT handlers—see the discussion of ABORT interrupts later in this section).

Figure 5-2 shows an example that illustrates how this works. Main program segment A includes an ON statement for SRQs. Subprogram B contains another ON statement for the same SRQ condition. When the main segment (A) calls subprogram B, the new definition takes effect. Then, B calls user-defined function C. Segment C does not contain an ON statement for the SRQ condition, so the handler

specifed by segment B still applies. However, when control returns to segment A after B and C are complete, the original SRQ handler definition automatically takes effect again, cancelling the handler link from segment B.

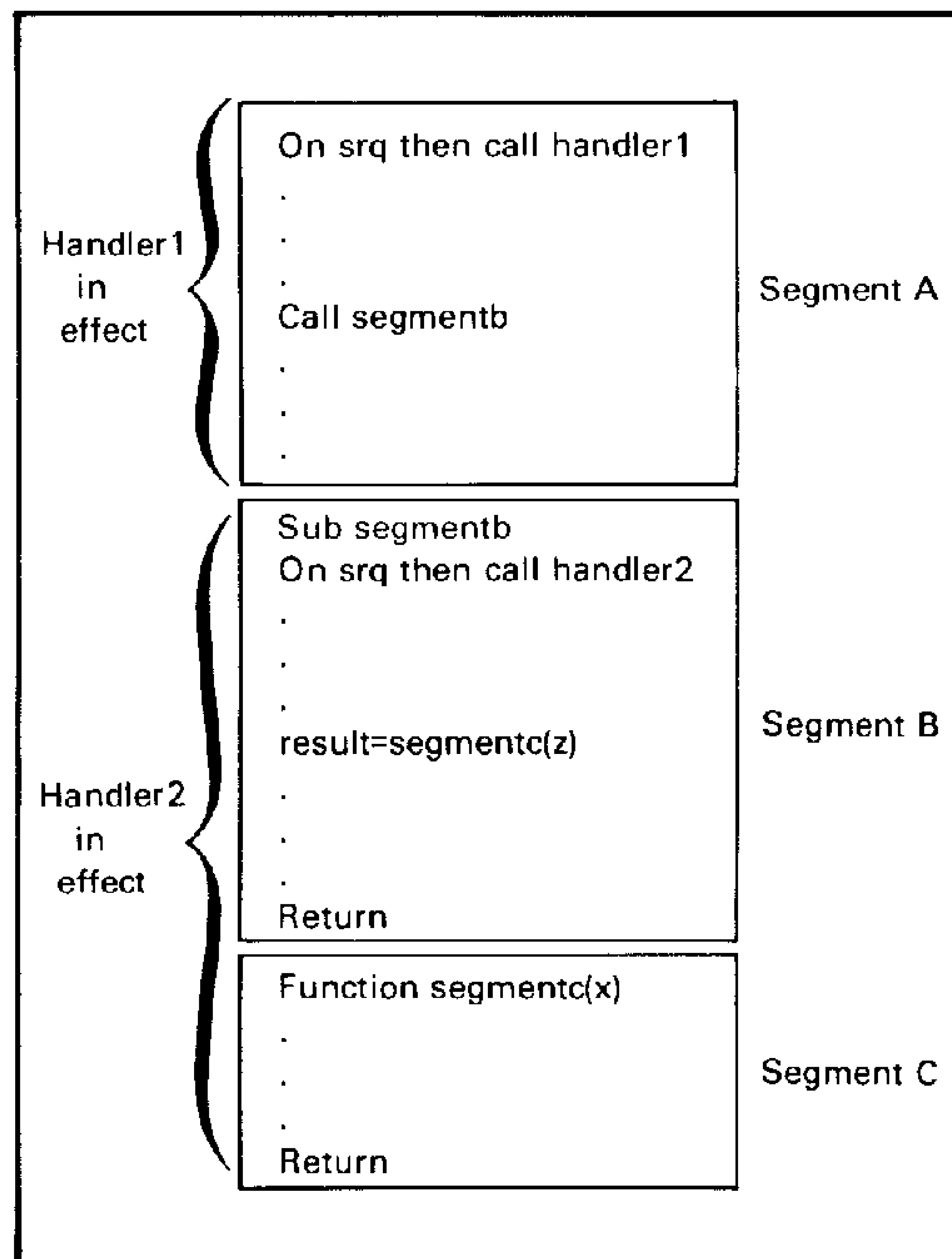


Fig. 5-2. *Subprograms and user-defined functions can link different handlers to a condition. When the subprogram or function returns to its caller, the caller's handler link is automatically restored (except for ABORT handlers).*

When handlers are linked in the main program segment with an ON...CALL statement, the handler definition is valid in all the segments called by the main program. However, if an ON...GOSUB statement defines a handler in a subprogram or user-defined function, the handler link is valid only within that segment.

An Example Handler

Figure 5-3 shows typical ON and ENABLE statements and handlers for an SRQ condition and user-definable key number 1. The first handler, called HANSRQ, processes SRQ interrupts from a GPIB device. This handler is linked in the main program and is written in the form of a subprogram (CALL). The second handler, called HANKEY1, processes interrupts generated when user-definable key number 1 is pressed. The handler is linked in a subprogram and is written in the form of a subroutine (GOSUB). As a result, it is defined only within the segment that links it.

```
On srq(10) then call hansrq
Enable srq
.
.
.
Call subprog1

Sub hansrq
Poll priadd,secadd;10
.
.
.
Resume

Sub subprog1
On key(1) then gosub hankey1
Enable keys
.
.
.
hankey1: rem ** User-definable key #1 handler **
Resume
Return
```

Fig. 5-3. *An example of linking and enabling handlers for an SRQ condition and for user-definable key number 1. Since the key handler (HANKEY1) is written as a GOSUB-type handler, it is defined only within the program segment that defines it.*

GPIB Interrupts

Seven different conditions on the GPIB can generate interrupts in the 4041. All of these conditions are linked to their handlers with ON statements and enabled with ENABLE statements. All GPIB conditions are disabled by default at power-up.

All of the conditions have some restrictions on when they can be used. For example, the EOI condition is only recognized when the 4041 is the controller-in-charge but is not participating (talking or listening) in the current data transfer. Table 5-3 summarizes the requirements for recognizing each GPIB interrupt.

The GPIB conditions may be specified in an ON statement with or without a LUN. If a LUN is specified in the ON statement, the interrupt is

recognized when the condition occurs on the port associated with the specified LUN. If a LUN is not specified, the interrupt is recognized when the condition occurs on the default port. At power-up, the default port is the standard port (GPIB0). The default port can be changed with the SELECT statement.

TABLE 5-3
GPIB INTERRUPT CONDITION REQUIREMENTS

Condition	Requirement
SRQ	Must be controller-in-charge.
EOI	Must be controller-in-charge, but not talking or listening.
IFC	Must be controller-in-charge, but not the system controller.
DCL	Must not be controller-in-charge.
MTA	Must not be controller-in-charge.
MLA	Must not be controller-in-charge.
TCT	Must not be controller-in-charge.

SRQ Interrupts

Most GPIB devices are capable of requesting service from the controller-in-charge by asserting the SRQ (Service Request) line on the GPIB. A device may assert SRQ for a variety of reasons. For example, Tektronix instruments assert SRQ if they receive a command over the bus that they do not understand or cannot execute. They may also assert SRQ when an operation is completed.

Whatever the reason, if an instrument asserts SRQ and a handler has been linked, the 4041 calls the handler routine. At this point, the 4041 doesn't know which device asserted SRQ, because all devices on the bus share the SRQ line. The SRQ line can be thought of as a simple parallel circuit (Fig. 5-4). When any device on the bus asserts SRQ, the line is asserted (pulled low).

Polling the devices. The first thing most SRQ handler routines do is find out which device is requesting service. That's where the serial poll comes in. The serial poll does just what it implies. It reads a status byte from each device on the currently SELECTed GPIB port whose address is listed in the POLL statement (or all devices, if no address list is included in the POLL statement).

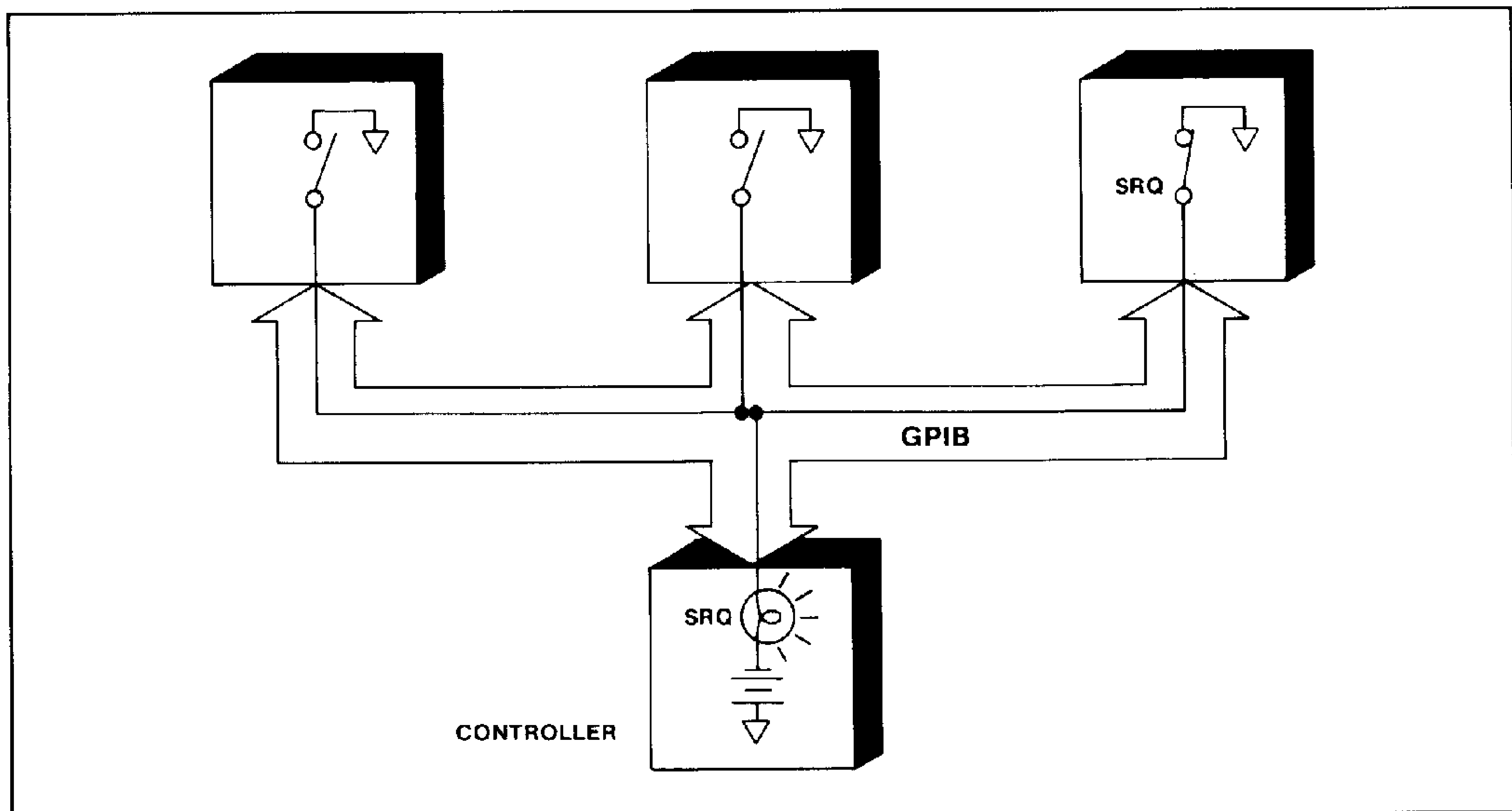


Fig. 5-4. *The GPIB SRQ line can be thought of as a simple parallel circuit. When any device on the bus requests service, the SRQ line is asserted, but the controller can't tell immediately which device is responsible for the interrupt.*

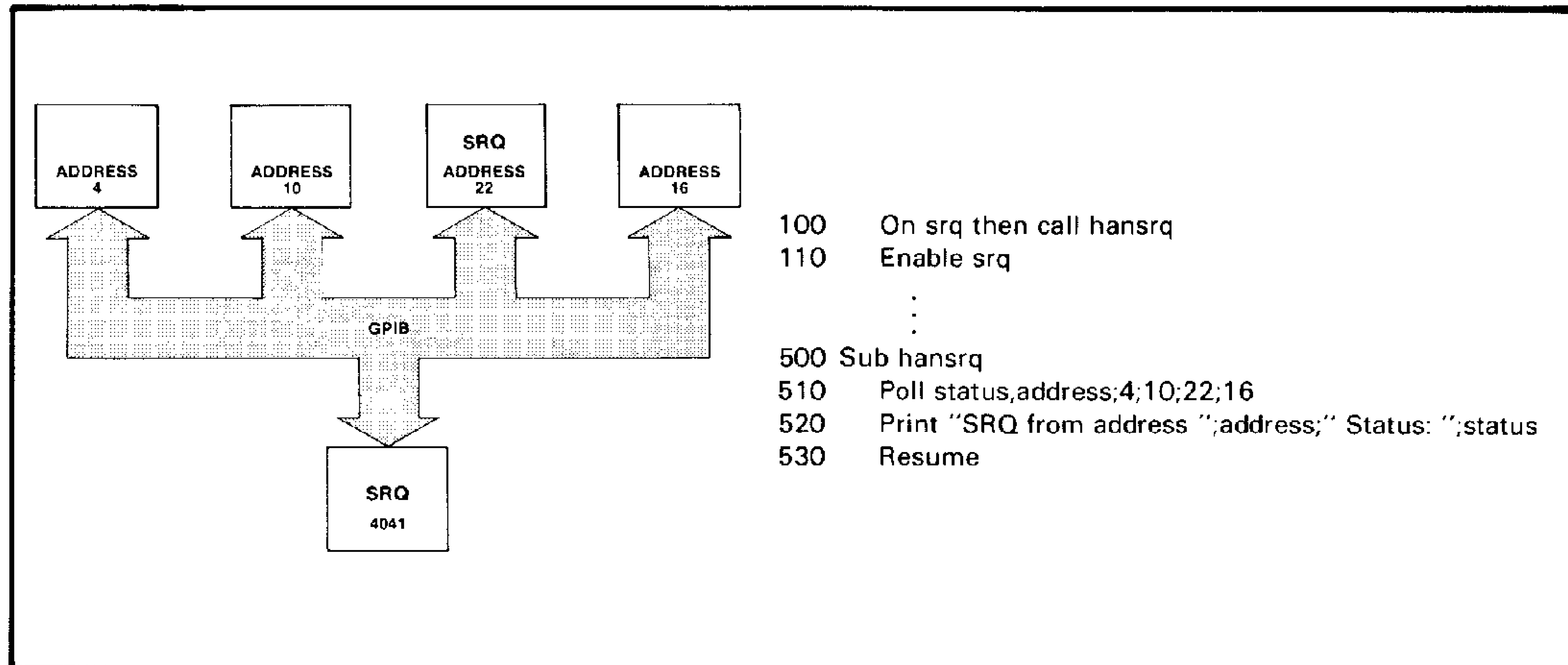


Fig. 5-5. *When a device asserts SRQ, the handler is called. The first thing the handler usually does is poll the instruments to find the device that is asserting SRQ and read its status.*

Each instrument reports a single status byte that indicates, among other things, if it is asserting SRQ and if so, why. The poll process stops with the first instrument found asserting SRQ. The address of this instrument and its status are returned through variables in the POLL statement.

For example, Fig. 5-5 shows a system with one device asserting SRQ. The 4041 calls the SRQ handler and conducts a serial poll of the devices on the SELECTed GPIB port. A segment of the handler program is shown in the figure. The POLL begins by reading the status of the device at address 4. Since this device is not asserting SRQ, the next device in the list is polled. This process continues until the device at address 22 is polled. This is the device that is asserting SRQ (indicated by setting a bit in its status byte). The poll process stops here and the status and address of this device are returned in the variables STATUS and ADDRESS, respectively.

POLL statement forms. The POLL statement can be used in three basic forms. In the first form, POLL gets the status of one particular instrument. It polls only the specified device and returns its status whether or not it is asserting SRQ.

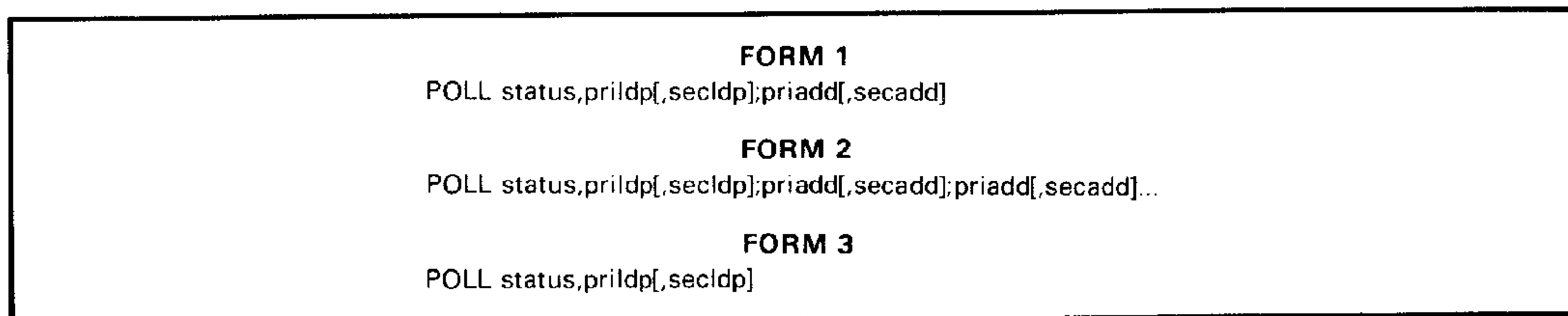
In the second form, a list of addresses is included in the POLL statement. The 4041 polls each instrument in the order that the addresses are listed. The poll process stops when the first device on the bus is found that is not asserting SRQ. If no device is asserting SRQ when this form of the POLL statement is executed, an error is generated.

The third form of the POLL statement includes no addresses. The 4041 polls all possible devices by beginning with primary address 0 and continuing through primary address 30. If no device in this range is found asserting SRQ, the same process is repeated trying every combination of secondary addresses. If SRQ is not asserted when the statement is executed, the 4041 generates an error. Figure 5-6 shows the three forms of the POLL statement.

SELECTing a GPIB port for POLL. The SELECT statement specifies which port a POLL will be conducted on. The power-up default is GPIB0, but the optional GPIB1 port can be selected at any time. The SELECT statement is described in Section 4.

A single SRQ handler can be used to handle SRQs from both GPIB ports by SELECTing the appropriate bus before conducting the POLL. The ASK$("SELECT") function returns the stream spec for the currently selected port.

The SPE parameter and POLL. The GPIB SPE parameter specifies the amount of time that the 4041 will wait for a response from a GPIB device when a serial poll is in progress. The POLL statement uses the value of the SPE parameter for the SELECTed stream spec. If an SPE value other than 10 milliseconds (the default) is to be used, the SPE value must be specified in a SELECT statement.



FORM 1
POLL status,prildp[,secldp];priadd[,secadd]

FORM 2
POLL status,prildp[,secldp];priadd[,secadd];priadd[,secadd]...

FORM 3
POLL status,prildp[,secldp]

Fig. 5-6. *The POLL statement can be executed in three basic forms. The first polls a specific instrument. The second polls a list of instruments, stopping with the first that is asserting SRQ, and the third polls all addresses, stopping with the first that is found asserting SRQ.*

Status byte format. Once the 4041 has read the instrument status and cleared the SRQ, the next question is: What does the status byte mean? With the exception of bit seven, the IEEE 488 standard doesn't say what the format of the status byte should be. Bit seven is designated by the IEEE 488 Standard as the service request bit. It is set when the instrument is asserting SRQ and cleared when the instrument is not asserting SRQ. The IEEE 488 standard doesn't define the rest of the bits.

However, the Tektronix Standard Codes and Formats more thoroughly defines the contents of the status byte while leaving room for many different devices to encode their particular status conditions. This makes status processing much simpler, since the format of the status byte is standardized. Figure 5-7 illustrates the Tek Codes and Formats status byte.

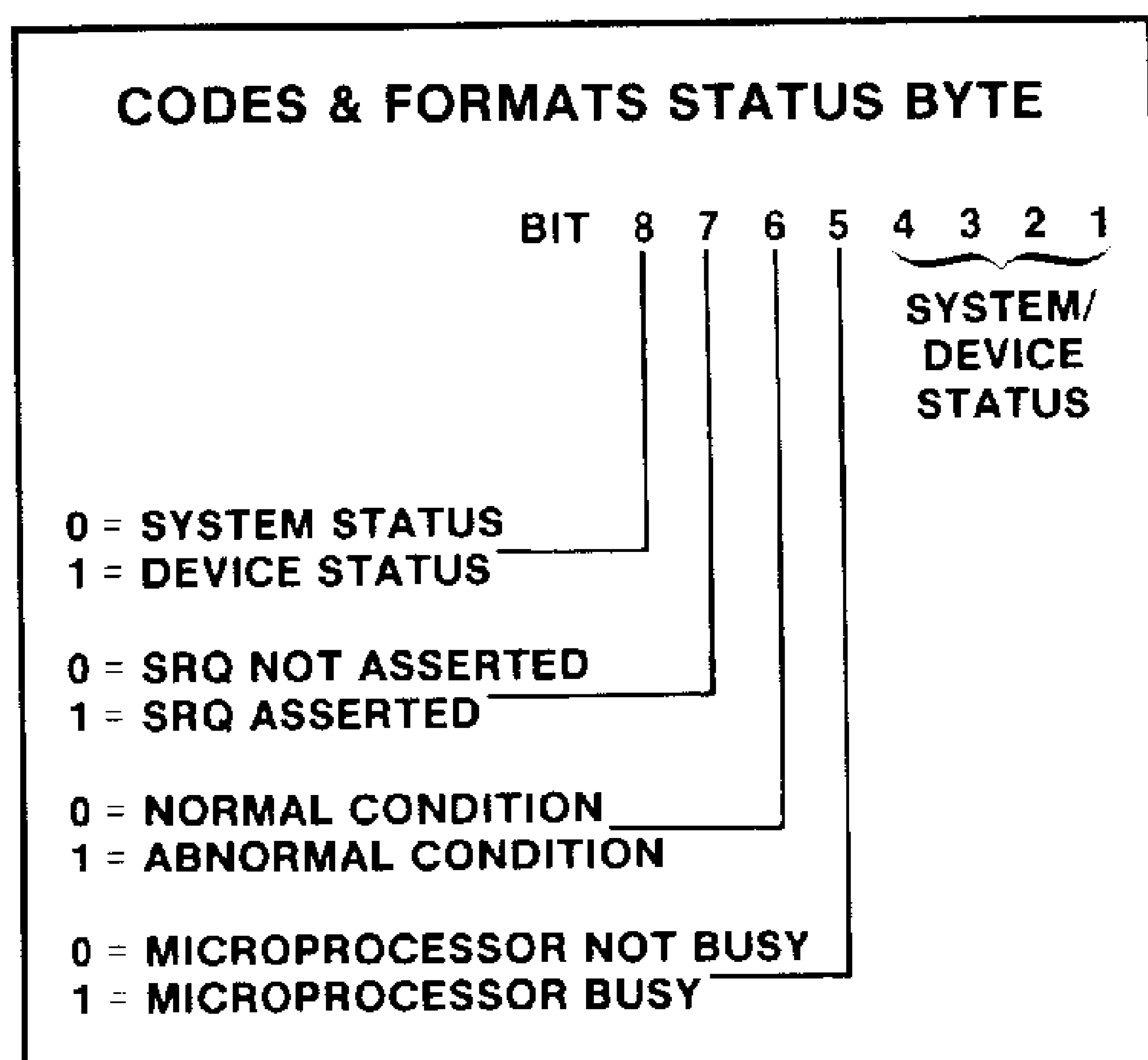


Fig. 5-7. *The Tektronix Standard Codes and Formats specifies a common format for status bytes. This makes processing status bytes from various Tektronix instruments much simpler.*

Bit eight of the status byte divides the status codes into two distinct classes—system status and device status. System status refers to codes that are common to all Tek Codes and Formats instruments. For example, all Tek Codes and Formats instruments report the power up condition using the same status code: 65.

Device status, on the other hand, refers to status bytes that are unique to the instrument type and refer to device-dependent functions or conditions. For example, the TEKTRONIX PS 5010 Power Supply reports a 197 status byte when its negative supply changes from current to voltage regulation. This status is defined only for the PS 5010 and similar instruments, so it's called a device status. System status bytes have bit eight clear; device status bytes have bit eight set.

Bit seven, as defined by the IEEE 488 standard, indicates whether the instrument is asserting SRQ or not. If bit seven is set, the instrument is asserting SRQ. If bit seven is clear, the instrument is not asserting SRQ.

Bit six differentiates between normal and abnormal conditions. Error and warning status bytes are considered abnormal status since they usually indicate a problem. A command error is an example of an abnormal condition. Status bytes that are reported as a normal part of operation are called normal condition status. Operation complete and power-up are normal condition status. Bit six is cleared for normal condition status bytes and it is set for abnormal conditions.

Bit five is the busy bit. It is set when the microprocessor in the instrument is busy and cleared when the microprocessor is not busy. If the microprocessor initiates an operation, such as a data acquisition, the busy bit is set only while the

microprocessor is involved in starting the operation. When the microprocessor's part of the job is complete, the busy bit is cleared. The instrument may be "busy" doing some operation, but as long as the microprocessor is not busy (ready to accept more command input), bit five is clear.

Bits one through four are used to encode the individual system or device status.

Figure 5-8 shows how all this fits together. A common status byte (power-up) is shown as an example. Notice that bit eight is clear, indicating that this is a system status byte—it is the same for all Tektronix instruments. Bit seven is set, so the instrument is asserting SRQ. Bit six is clear because power-up is a normal condition and bit five is clear because the microprocessor has completed the self-test and is no longer busy.

The last four bits of the status byte contain the code for power-up (0 0 0 1). The decimal value of this status byte is 65. This is the value returned in the status variable in the POLL statement.

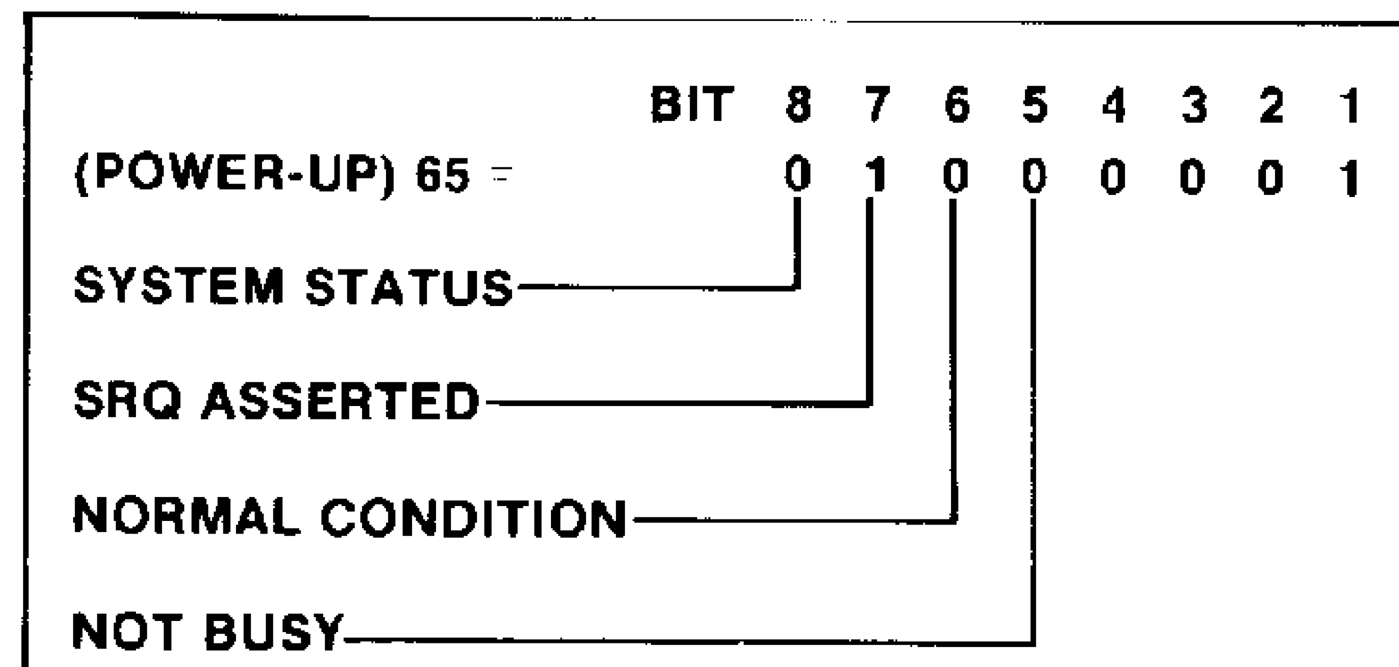


Fig. 5-8. *An example of a system status byte—power-up. This status byte is the same for all instruments that conform to the Tektronix Standard Codes and Formats.*

Processing the Status Byte

The consistent format of the Tektronix Standard Codes and Formats status byte makes processing the status byte much easier. For example, system status bytes (status bytes with bit eight clear) can be processed by common routines for all instruments since the status codes are the same for all instruments that conform to the Standard Codes and Formats. Device status bytes (bytes with bit eight set) are processed by separate routines for each device type.

```
100     Dim priadd(4)
110     priadd(1)=4
120     priadd(2)=10
130     priadd(3)=22
140     priadd(4)=16
150     Poll status,address;priadd(1);priadd(2);priadd(3);priadd(4)
160     If status>=128 then goto devsta
170     Rem ** system status processing begins here **
           :
300     Rem ** device status processing begins here **
310 Devsta: For i=1 to 4
320     If address=priadd(i) then exit to devpro
330     Next i
340 Devpro: Goto I of devpro1,devpro2,devpro3,devpro2
350 Devpro1: rem ** processing for device #1 begins here **
           :
400 Devpro2: rem ** processing for devices #2 and #4 begins here **
           :
500 Devpro3: rem ** processing for device #3 begins here **
           :
```

Fig. 5-9. *Device and system status bytes can be separated by comparing them to 128. System status bytes can be processed by a common routine, while device status byte must be processed by individual routines for each instrument type.*

Differentiating system and device status. Differentiating between system and device status is simple. System status bytes all have values less than 128 (bit eight clear) and device status bytes all have values greater than or equal to 128 (bit eight set). After the POLL process is complete, the handler can compare the status value to 128. If it is less than 128, the common system status processing routine can be called. Otherwise, the address returned in the POLL statement can be used to call a routine that handles the device-specific codes generated by this instrument. Figure 5-9 shows how this technique might be implemented in 4041 BASIC.

In this program, the status value returned by the POLL statement is compared to 128. Codes less than 128 are processed by the system status (SYSSTA) routine. Codes that are greater than or equal to 128 are processed by separate routines for each instrument type. The instrument address returned by POLL is compared to a list of addresses. When a match is found, the position of the address in the table is used as an index to a computed GOSUB that calls the appropriate routine.

Reporting errors. Reporting an error message is one of the common tasks for an SRQ handler. There are a variety of ways to report a message based on the status value. One simple way is to take advantage of the 4041's string array capability (Fig. 5-10). The status value is converted to an array index and used to select a message. The selected message is printed on the console device.

```
100   Dim message$ to 6
110   Message$(1)="Command Error"
120   Message$(2)="Execution Error"
130   Message$(3)="Internal Error"
140   Message$(4)="Power Fail"
150   Message$(5)="Execution Warning"
160   Message$(6)="Internal Warning"
170   On srq then gosub polsrq
180   Enable srq
        :
1000 Polsrq: Poll status,address;10
1010   If status>=128 then goto devsta
1020   Print message$(status-96)
1030   resume
1040 devsta: rem ** device status processing **
        :
```

Fig. 5-10. *Error messages can be stored in a 4041 string array. In this program the status value is used to select the appropriate error message. The error message is printed on the console device.*

Most Tektronix instruments also implement an error query (ERR?) that returns more specific information about the nature of an error. For example, if an instrument reports a command error status, the ERR? query returns a numeric code that indicates more specifically what was wrong with the command that caused the error. The command may have omitted a required argument. The error code will indicate this.

```
100   Dim message$ to 6
110   Message$(1)="Command Error"
120   Message$(2)="Execution Error"
130   Message$(3)="Internal Error"
140   Message$(4)="Power Fail"
150   Message$(5)="Execution Warning"
160   Message$(6)="Internal Warning"
170   On srq then gosub polsrq
180   Enable srq
        :
1000   Poll status,address;10
1010   If status band 32=0 then goto normsta
1020   Print "Abnormal condition status: ";status;" from address ";address
1030 normsta: resume ! Ignore normal status conditions
```

Fig. 5-11. *The status reporting program can be modified to report only abnormal condition status by testing bit six of the status byte with the BAND operator. Normal condition system status is ignored.*

Normal and abnormal condition status. Some of the conditions reported in the previous program, such as power-up, are normal conditions that are usually of no concern to the operator. You may want to report only abnormal conditions to the operator and either ignore or take other action on normal conditions. Bit six of the Tektronix Standard Codes and Formats status byte can be used to differentiate between normal and abnormal conditions. When bit six is set, the status byte represents an abnormal condition, such as a command error. When bit six is clear, the status value represents a normal condition.

The program in Fig. 5-11 uses the binary operators to test bit six of the status byte. The BAND (Binary AND) operator does an AND operation on the two values and returns the result in the variable on the left side of the equal sign. In this case, the value is ANDed with the value 32. Since this value only has bit six set, the result of the BAND operation will be 32 if bit six is set in the status byte, or zero if bit six is clear.

Error logging. Errors (abnormal conditions) may also be logged to a system error log file in addition to (or instead of) reporting them on the console device. Figure 5-12 shows a program that logs abnormal condition status bytes in a file called ERRLOG. Each time an error occurs, a message is printed in the log file. The message includes the address of the instrument that generated the SRQ, the identification string returned from an ID? query, the error code returned from an ERR? query, and the time and date that the error occurred. The instrument settings (read with a SET? query) could also be included in the log.

```
100    Open #40:"errlog(open=update,clip=yes)"
110    On srq then call pollsrq
120    Enable srq(3)
         :
         :
1000 Pollsrq: Poll status,address;10
1010 If status band 32=0 then goto normsta
1020    Input #address prompt "ID?":ident$
1030    Input #address prompt "ERR?":err$
1040    Print #40:"Abnormal status from device ";ident$;
1050    Print #40:" Bus address: ";address
1060    Print #40:"Status= ";status;" Error code= ";err$
1070    Print #40:"Error occurred at: ";ask$("time")
1080 Normsta: resume ! Ignore normal condition status
```

Fig. 5-12. *Errors (indicated by abnormal condition status) can be logged to an error log file as shown in this program. The program logs the instrument ID, error code, bus address, status byte value, and the date and time the error occurred.*

```
100    Open #3:"GPIB0(pri=3,sec=0):"
110    Print #3:"WRI ON"
120    On srq(3) then call pollsrq
130    Enable srq(3)
140    Integer waveform(2048)
         :
         :
300    Print #3:"ARM A"
         :
         :
1000 Pollsrq: Poll status,address;3,0
1010   If (status band 163)<>131 then goto done
1020   Input #3 prompt "READ A" using "+8%":waveform
1030   Input #3:term !Get the semicolon terminator
1040     dataflag=1
1050 done: resume
```

Fig. 5-13. *The Operation Complete or Waveform Readable features of some instruments can be used to make data acquisition more efficient. Instead of wasting time waiting for an acquisition to complete, the program continues other processing until the WRI interrupt occurs. The SRQ handler reads the data and sets a flag to indicate that new data is available.*

Using SRQ interrupt to control program flow. Some SRQ interrupts can be used to more efficiently control program flow. For example, a system might use the Operation Complete (OPC) interrupt to detect the end of a data acquisition. Instead of wasting time waiting for an acquisition to complete, the program can continue with other processing. When the instrument asserts SRQ and reports Operation Complete status, the handler can read the data.

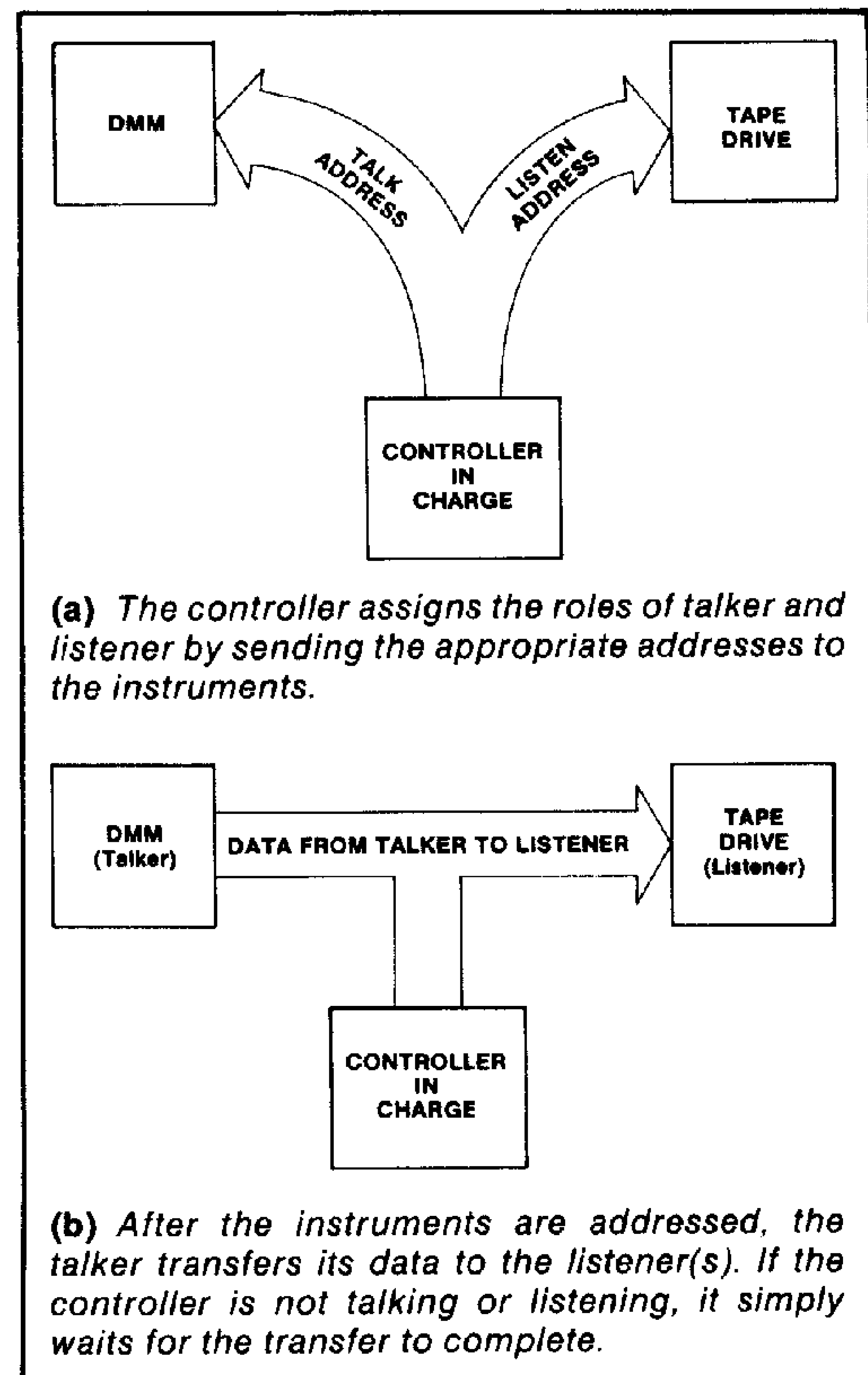
Figure 5-13 shows a program that uses this technique to acquire data from a 7612D Programmable Digitizer. The program enables the Waveform Readable Interrupt (WRI) and initiates an acquisition. Then, the program continues with other processing. When the 7612D asserts SRQ, the handler is called. If the 7612D reports waveform readable status for channel A, the handler reads the waveform data. A flag is set to tell the main program that new data is available and the handler executes a RESUME to resume main program execution.

EOI Interrupts

The End Or Identify line on the GPIB may be used to terminate a device-dependent message transfer. When the 4041 is the controller-in-charge, but not participating in a data transfer (talking or listening), it can detect the EOI at the end of a data transfer. If a handler is linked for the EOI condition and the EOI condition is enabled, the 4041 generates an interrupt when EOI is asserted.

Handlers for the EOI condition are linked with an ON EOI statement similar to the ON SRQ statement. The exit, enable, and disable conditions are the same for all GPIB conditions and are listed in Table 5-1. If a LUN is specified in the ON EOI(lun) statement, the interupt is recognized when EOI is asserted on the port associated with that LUN.

The EOI interrupt can be used to monitor data transfers that do not involve the 4041 as a talker or listener. For example, it is usually faster to transfer data directly from a digitizer to tape drive instead of reading the data into the controller and then writing it out to the tape drive. In this case, the 4041 is not directly involved as a talker or listener in the data transfer, though it does address and unaddress the talker and listeners (Fig. 5-14).



(a) *The controller assigns the roles of talker and listener by sending the appropriate addresses to the instruments.*

(b) *After the instruments are addressed, the talker transfers its data to the listener(s). If the controller is not talking or listening, it simply waits for the transfer to complete.*

Fig. 5-14. *The controller-in-charge assigns the roles of talker and listener to devices in the system. The controller may or may not be involved in the data transfer.*

A program that implements this idea is shown in Fig. 5-15. The 4041 begins by linking the EOI interrupt handler routine with an ON statement and enabling the EOI condition. Next, the program addresses the digitizer to talk and addresses the tape drive to listen. If the digitizer requires a send data command, such as CURVE?, this command must be sent first. When the addressing sequence is complete, the 4041's involvement in the data transfer is finished. The digitizer begins sending data to the tape drive and the 4041 waits at line 180. The 4041 could perform other processing during the transfer instead of just waiting.

```
100    Open #1:"GPIB0:"
110    On eoi(1) then gosub 190
120    Enable eoi(1)
130    Digpri=4
140    Tappri=10
150    Listen=32
160    Talk=64
170    Wbyte atn(digpri+talk,tappri+listen)
180    Wait
190    Wbyte atn(unt,unl)
200    Branch 210
210    .
       :
```

Fig. 5-15. *The EOI interrupt can be used to monitor data transmissions between two GPIB devices without involving the 4041 in the transfer. The 4041 addresses one device to talk and one or more to listen. Then, it waits for EOI to be asserted, indicating the end of the transfer.*

At the end of the transfer, the digitizer asserts EOI with the last byte it sends. The 4041 detects the EOI and calls the handler. In this example, the handler is simply a statement that unaddresses the instruments using the WBYTE statement. The BRANCH statement terminates the handler and continues execution at the next line.

The EOI interrupt condition is recognized only when the 4041 is the controller-in-charge but not talking or listening.

IFC Interrupts

The Interface Clear (IFC) line on the GPIB is used to reset the interfaces of all devices on the bus to a known state. When IFC is asserted, all devices on the bus are unaddressed and, if the system controller is not the current controller-in-charge, the system controller reverts to being the controller-in-charge.

The IFC interrupt condition can be used when the 4041 is **not** the system controller but it **is** the controller-in-charge. When IFC is asserted, the 4041 can recognize the condition and perform any necessary processing for returning control to the system controller. Only the system controller can assert IFC.

If IFC is asserted when the 4041 is not the controller-in-charge, the GPIB interfaces are unaddressed, but an IFC interrupt is not generated.

If a LUN is specified in the ON IFC(lun) statement, the interrupt condition is recognized when IFC is asserted on the port associated with the specified LUN.

Be careful not to confuse the IFC interrupt condition with the WBYTE IFC function. WBYTE IFC asserts IFC when the 4041 is the system controller. The IFC interrupt condition is used to recognize IFC when the 4041 is not the system controller but it is the controller-in-charge.

DCL Interrupts

The 4041 can also generate an interrupt when it receives the Device Clear (DCL) interface message or when it is listen addressed and it receives the Selected Device Clear (SDC) interface message. When either DCL or SDC are received, the 4041 aborts any data transfer in progress and calls the DCL handler (if one is linked). The device clear handler can be used to initialize a program or variables as required by the application program.

For example, a DCL interrupt handler might initialize all data variables in a program to zero, reset flags, and set 4041 parameters (such as key handler links) to an initialized state.

The DCL interrupt condition is only recognized when the 4041 is not the controller-in-charge. If a LUN is specified in the ON DCL(lun) statement, the interrupt is recognized when a DCL or SDC is received on the port associated with the specified LUN.

MTA and MLA Interrupts

The 4041 can act as a talker or listener in a GPIB system when it is not the controller-in-charge. When the 4041 is acting as a talker/listener, the MTA and MLA interrupts can be used in a program to detect the receipt of the 4041's talk or listen address, respectively. This provides a convenient means of calling routines that send data when the 4041 is addressed to talk and calling routines that receive data when the 4041 is addressed to listen.

The MTA and MLA interrupt conditions are linked and enabled in the same fashion as other GPIB interrupt conditions. They are only recognized when the 4041 is not the controller-in-charge.

Figure 5-16 illustrates a system that uses the 4041 as a node controller for a large GPIB system. The

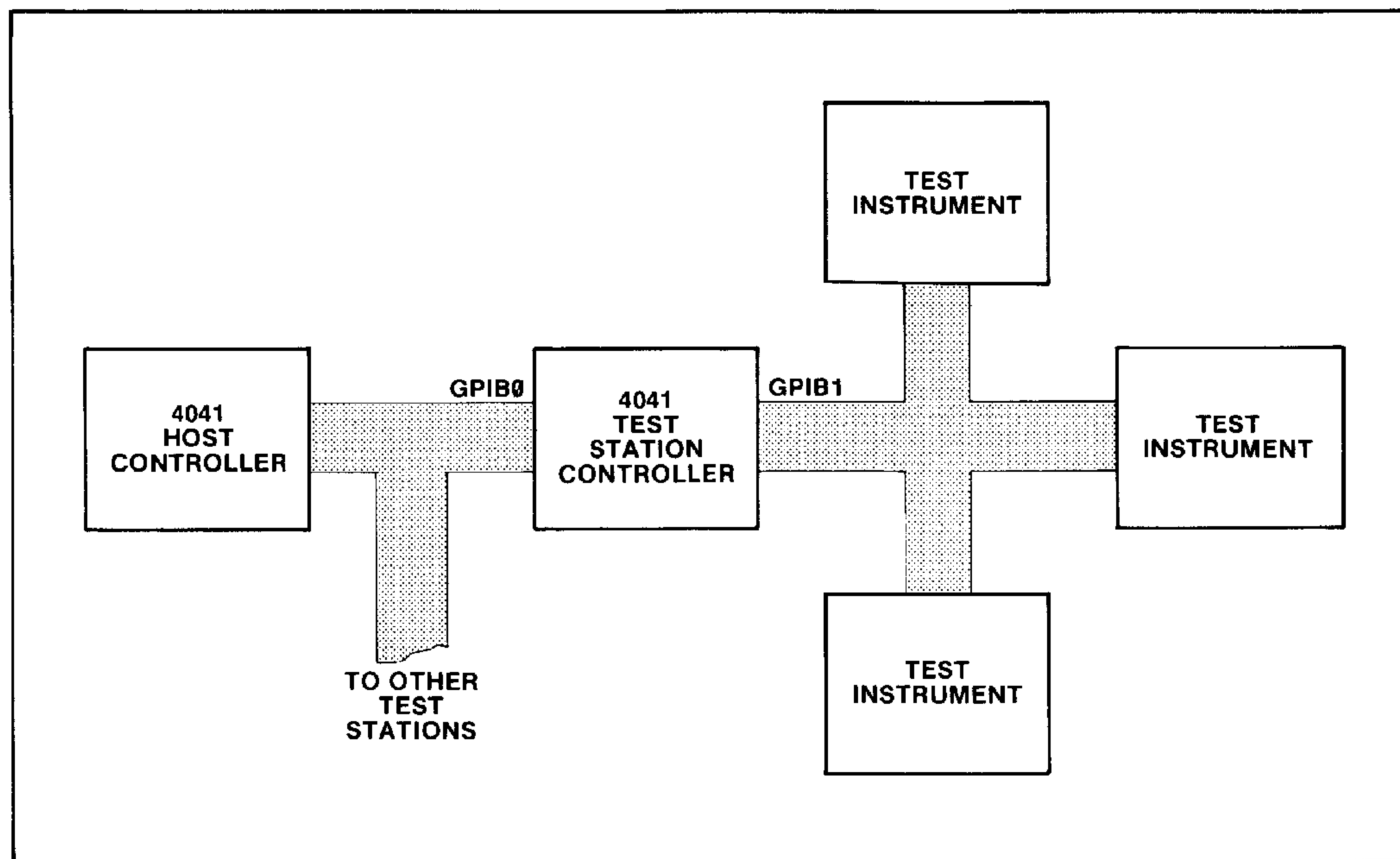


Fig. 5-16. *The 4041 can act as a system node controller. On one GPIB port, it operates as a talker/listener using the MTA and MLA interrupts to drive the program. On the other GPIB port, the 4041 acts as the system controller for a group of test instruments.*

4041 acts as a talker/listener on the standard (GPIB0) port. This port is used to communicate with the host controller (another 4041). The second GPIB port (GPIB1) is used to control a local system of instruments making measurements.

Each local system performs as a stand-alone automated test system, making measurements, performing tests, and logging results to a file. The host controller collects test result data from each of the node controllers and produces summary reports that indicate failure modes and failure rates for the entire system as well as individual test stations. When the host controller requests results from a test station, it addresses the node controller to listen and sends a command that indicates the type of data being requested. The command might be "FAILURE?."

The local 4041 detects the listen address and generates an MLA interrupt. The MLA handler routine decodes the FAILURE? command and prepares to send the requested information to the host controller. When the failure data is ready for transmission, the node controller can assert SRQ using the WBYTE SRQ(status) function. The host controller recognizes this SRQ, polls the node controller, and reads its status byte. The status byte tells the host controller that the failure data it requested is ready for transmission.

In response, the host controller addresses the node controller to talk. The 4041 recognizes the talk address and generates an MTA interrupt, which initiates the transfer of the failure data.

Figure 5-17 shows the MTA and MLA interrupt handler segment of a program that would accomplish this function. Notice that when the local 4041 node controller is not interacting with the host controller via the GPIB0 port, it is performing normal system controller functions with the local devices on the second GPIB port.

```
100    Set driver "GPIB0(sc=no,ma=16):"
110    Open #1:"GPIB0:"
120    On mta(1) then gosub talk
130    On mla(1) then gosub listen
140    Enable mta(1),mla(1)
150    Rem ** From here, the program procedes with  **
160    Rem ** normal system controller tasks for    **
170    Rem ** the local test system. This task      **
180    Rem ** continues until the host 4041         **
190    Rem ** addresses this 4041 to talk or listen. **
         :
1000 Talk: Print #1:response$  ! send the response to the host
         :
2000 Listen: Input #1:command$  ! accept the command from the host
         :
```

Fig. 5-17. *This program segment illustrates how the MTA and MLA interrupts might be used to drive a program that interacts with a host controller over the GPIB. The local 4041 acts as a talker/listener on this port and as a normal system controller on the other port.*

TCT Interrupts

The Take Control (TCT) interface message is used to transfer control to another device. The controller-in-charge passes control by addressing the new controller to talk and sending the TCT message. When the 4041 is operating in a multiple-controller system and it is not the current controller-in-charge, it can recognize the TCT message as an interrupt condition. The handler can prepare the application program to take control of the bus.

The TCT condition is enabled and linked in the same fashion as other GPIB functions. If a logical unit number is specified in the ON statement, the interrupt is recognized when the 4041 is addressed to talk and receives the TCT message on the port associated with that LUN. The TCT condition is not recognized if the 4041 is the controller-in-charge.

ABORT Interrupts

When a user presses the ABORT key on the 4041 front-panel or program development keyboard, or when a control-C is received from the COMM port used as a console device, the 4041 generates an ABORT interrupt. ABORT interrupts are handled by a default system handler unless an ON ABORT statement is executed to link a user-written ABORT handler. The system handler simply halts program execution and prints the SYSTEM ABORTED message on the console device.

ABORT interrupts can be disabled with the DISABLE ABORT statement. When disabled, abort interrupts are not recognized even if a user-written handler is linked. The ABORT condition can be re-enabled with ENABLE ABORT. The default condition is enabled.

Nested ABORT handlers. ABORT handlers are a little different than other interrupt conditions when it comes to nesting handlers. ABORT handlers do not "stack" like other handlers do.

A handler called with an ON ABORT CALL statement is effective for all subprograms or subroutines called by the segment that includes the ON ABORT statement. However, if a subprogram or subroutine links a new handler, the previous link is lost. Returning from the subprogram does not automatically reinstate the previous link. Returning from a subprogram that defines an ABORT handler returns the ABORT link to the system handler.

A handler called with an ON ABORT GOSUB statement is effective only within the subroutine that links it.

```
100    On abort then call abohan
110    Print "MAIN SEGMENT EXECUTING"
120    For i=1 to 1000
130     Next i
140    Call segmentb
150    Print "MAIN SEGMENT EXECUTING AGAIN"
160    For i=1 to 1000
170     Next i
180    Print "EXECUTION COMPLETE"
200 Quit: End
300 Sub segmentb
310    Print "SEGMENT B EXECUTING""
320    For i=1 to 1000
330     Next i
340    Call segmentc
350    Return
360    End
400 Sub segmentc
410    On abort then gosub abohan2
420    Print "SEGMENT C EXECUTING""
430    For i=1 to 1000
440     Next i
450    Return
460 Abohan2:  print "ABORT IN SEGMENT C"
470    Branch quit
480    End
500 Sub abohan
510    Print "ABORT IN MAIN OR SEGMENT B"
520    Branch quit
530    End
```

(a) *This program illustrates the use of ABORT handlers in subprograms.*

```
*run
MAIN SEGMENT EXECUTING
(ABORT pressed)
ABORT IN MAIN OR SEGMENT B

*run
MAIN SEGMENT EXECUTING
SEGMENT B EXECUTING
(ABORT pressed)
ABORT IN MAIN OR SEGMENT B

*run
MAIN SEGMENT EXECUTING
SEGMENT B EXECUTING
SEGMENT C EXECUTING
(ABORT pressed)
ABORT IN SEGMENT C

*run
MAIN SEGMENT EXECUTING
SEGMENT B EXECUTING
SEGMENT C EXECUTING
MAIN SEGMENT EXECUTING AGAIN
(ABORT pressed)

SYSTEM ABORTED LINE # 170
```

(b) *If the program is aborted while the main segment is executing or while subprograms B or C are executing, the user-define ABORT handler is called. However, after subprogram C returns, the system handler is linked again.*

Fig. 5-18. *Abort handler links are not "stacked" like other handler links. When a subprogram returns to it's caller, the previous links are not restored. Instead, the system handler is re-linked.*

Part **a** of Fig. 5-18 shows a program that illustrates how nested ABORT handlers work. The main segment links a handler subprogram (ABOHAN) with the ON statement in line 100. Since this handler is in the subprogram form, the handler is effective for the main program as well as the subprograms and subroutines it calls. Thus, the handler is effective for both the main program and for the first subprogram (SEGMENTB). If the program is aborted while the main segment or segment B is executing, the ABOHAN handler is called. This handler prints the message "ABORT IN MAIN OR SEGMENT B" and stops execution.

Segment B calls a second subprogram (SEGMENTC). This subprogram defines a new ABORT handler in line 410. Since this handler is linked with a GOSUB-type ON statement, the handler is only effective in this subprogram. If the program is aborted while segment C is executing, the new handler (ABOHAN2) is called and the message "ABORT IN SEGMENT C" is printed. When segment C returns control to segment B, the original handler link is not restored. Instead, the system ABORT handler is linked. As a result, if the program is aborted after segment C is complete, the system abort handler is called and the SYSTEM ABORTED message is printed.

Part **b** of Fig. 5-18 shows the output from this program when it is aborted in various stages of execution.

ERROR Interrupts

If an error condition occurs while executing a program or in immediate mode, the 4041 calls a system error handler that reports the error number and, if the error occurred in a program, the line number where it occurred. The ON ERROR statement can be used to link user-written handlers to specified interrupt conditions.

Setting up error handlers. The ON ERROR statement is very similar to the ON statements for other conditions. Error handlers may be written as subprograms, called with an ON ERROR...CALL statement, or they may be written as subroutines, called with an ON ERROR...GOSUB statement.

In addition, the ON ERROR statement provides several options that allow you to link a handler for specific error numbers or errors that occur on a specific LUN or group of LUNs. In it's simplest form, the ON ERROR statement links a handler to one specified error condition. For example:

```
On error(80) then call errhan
```

tells the 4041 to call the handler ERRHAN when an error number 80 occurs. A group of errors may also be linked to a single handler by specifying the range of error numbers in the following form.

```
On error (80 to 100) then call errhan
```

This ON statement links all errors between 80 and 100, inclusive, to the ERRHAN handler.

Some errors, such as I/O errors, are related to specific LUNs. For these errors, a specific LUN or group of LUNs can be linked to a handler. The LUN numbers are specified after the error numbers, separated by a comma. For example:

```
On error(800 to 839,1 to 10) then call errhan
```

This statement links the ERRHAN handler to all error conditions in the range 800 to 839, inclusive, that occurs on logical unit numbers 1 through 10, inclusive.

If an error occurs that does not fit any of the conditions specified in the ON ERROR statements, the error is handled by the system handler.

ON ERROR Statement order. If several ON statements are executed and any of the conditions overlap, it is important to execute the ON statements in the correct order. The most "general" handlers (i.e., those that cover a wider range of errors or LUNs) should be linked first, followed by the more specific handlers (the handlers that deal with a narrower range of errors or LUNs). The following statements illustrate this concept:

```
On error(800 to 839) THEN CALL ERRHAN
On error(812,10) then call er812han
```

The first ON statement sets up a handler for all error conditions between 800 and 839, regardless of what LUN they occur on. The second statement sets up a handler for a specific error number on a specific LUN. Notice that the more specific handler is set up after the general handler is set up. If the order is reversed, the results are different.

```
On error(812,10) then call er812han
On error(800 to 839) then call errhan
```

In this case, the first ON statement sets up a handler for the specific error condition but the second statement overrides the first one because the specific error (812) is included in the range of the general handler. Thus, it is usually best to set up general handlers first, followed by more specific handlers.

Proceed Mode I/O Errors. When an I/O error occurs during a proceed mode I/O operation, the error is reported a little differently. All proceed mode I/O errors are reported as error number 999. The ASK$("ERROR") function can be used to get the actual error number and the line number of the statement that initiated the I/O operation in which the error occurred.

User-defined errors. 4041 BASIC includes a TRAP statement that allows the user to generate a "user-defined" error. The TRAP statement generates an error number 1000 which can be linked like any other error with an ON ERROR statement. The TRAP error can be used in any convenient manner for the application program.

The ASK$("ERROR") function. The ASK$("ERROR") function returns more detailed information about the nature of an error. It returns four values (eight for a proceed mode I/O error) in a string variable. The first value is the error number. For proceed mode errors, this value is always 999. The second value is the line number in which the error occurred. For proceed mode errors, this value is zero. The third value is the logical unit number related to the error. If the error is not related to a LUN, this value is -1. For proceed mode errors, this value is zero. The third value contains a count of

how many consecutive times the error occurred without another intervening error. For proceed mode errors, this value is also zero.

Proceed mode errors return four more values. The first of these values is the error code that describes the specific type of error. The general error code for all proceed mode I/O errors is 999, but this value indicates the specific type of error that occurred. The second value is the line number of the proceed mode I/O statement that initiated the I/O in which the error occurred. The third value is the logical unit number on which the error occurred, and the fourth value is the error repetition count.

OFF ERROR statements. The OFF ERROR statement cancels the effect of any ON ERROR statement created in the current program segment whose arguments exactly match the arguments of the OFF statement.

If the arguments of an OFF ERROR statement do not exactly match any ON ERROR statements in this program segment, the OFF statement is ignored. When an OFF statement is executed, the error conditions specified in the OFF statement are handled either by another handler that is also linked to this condition or to the system handler if no other user-written handler is presently in effect for this condition.

IODONE Interrupts

When the 4041 completes a proceed-mode I/O operation, it can generate an interrupt. This interrupt is automatically used by 4041 BASIC to coordinate proceed mode I/O, but it can also be linked to a user-written handler. The IODONE interrupt is automatically enabled when proceed mode I/O is enabled. The interrupt is automatically disabled when proceed mode is disabled.

Handlers for the IODONE condition are set up with an ON IODONE statement. If a LUN is specified in the ON IODONE statement, the handler is called when proceed mode I/O completes on the specified LUN. If no LUN is specified, the handler is called when proceed mode I/O completes on the console device.

The OFF IODONE statement cancels links set up with the ON IODONE statement.

KEY Interrupts

The front-panel numeric keys on the 4041 can be used as user-definable keys. In addition, when the console device is a terminal attached to one of the COMM ports, the user-definable keys can be emulated on the terminal. Function keys 1 through 10 are emulated by pressing control-F followed by a corresponding digit from 0-9. Function keys 11-20 are emulated by pressing control-D followed by a corresponding digit from 0-9.

When the user-definable keys are enabled with the ENABLE KEYS statement, pressing a user-definable key causes an interrupt. The user-definable keys are all enabled or disabled together. They can't be enabled or disabled individually. However, a handler need not be be defined for all keys. The user-definable keys do not have pre-linked system handlers. However, the ON KEY statement can be used to link a user-written handler to a specified key. For example, the statement:

```
On key(1) then call key1han
```

calls the handler subprogram KEY1HAN when user-definable key 1 is pressed on the front-panel or when a control-F followed by a 1 is received from a terminal used as the console device.

Keys that do not have handlers linked are ignored.

User-definable keys and the console device. User-definable keys are recognized only on the console device. If the front-panel is the console device, only the front-panel numeric keys are recognized as user-definable keys. If the console device is assigned to one of the COMM ports, the front-panel keys are disabled and a control-F or control-D followed by a digit received from the COMM port is accepted as a user-definable key.

Key queueing. When user-definable keys are disabled, the 4041 remembers the last user-definable key pressed. If the keys are later enabled, the handler for that key is called (if one is linked). Since the keys are automatically disabled while a previous key interrupt is being processed, one key press can be "queued." As soon as the handler for the first key is complete, the keys are re-enabled and the handler for the second key is called.

The ASK("KEY") function. The ASK("KEY") function returns the number of the key awaiting service, if any. A value of zero indicates that no key is awaiting service. A handler can be set up for each individual key using the ON KEY(n) statement, where n is the key number.

Section 6—Processing and Displaying Data

Most GPIB systems make some type of measurement as part of their job, whether it be sensing a logic level, making a simple voltage measurement, or acquiring a complete waveform. The first step in making the measurement is to acquire the signal and convert it to digital information. That is the job of the acquisition instrument (e.g., waveform digitizer, DMM, etc.). The output of this instrument is then sent to the controller over the GPIB (Fig. 6-1).

Once data is acquired and transferred to the 4041, some processing is usually required to extract the desired information. It's important to remember that the 4041 is processing a digital representation of the input signal—a string of numbers—not the signal itself. In the case of a waveform digitizer, these numbers usually represent vertical signal amplitudes at discrete sample points along the signal. The position of each number in the series represents its time location on the signal (Fig. 6-2).

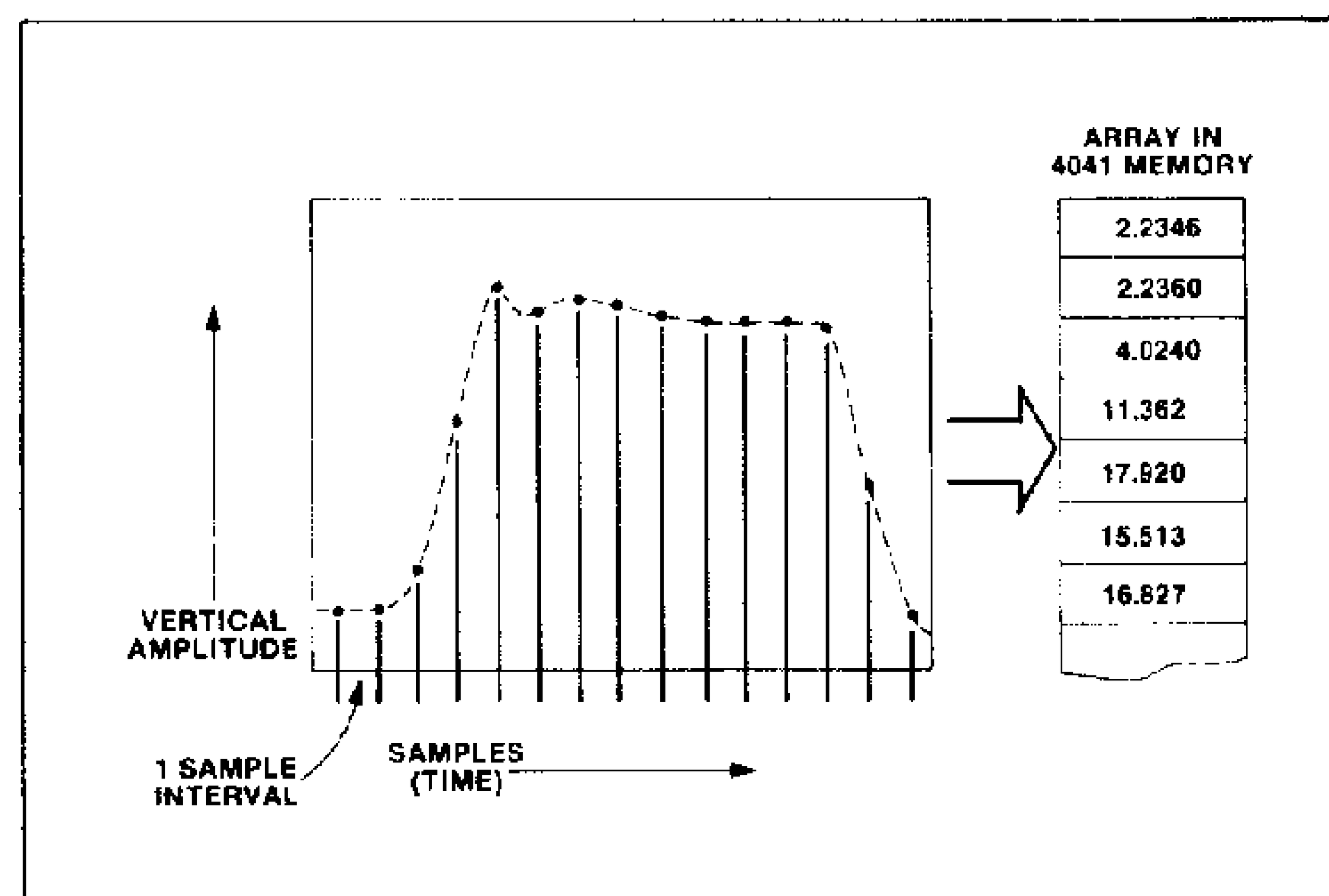


Fig. 6-2. *Samples collected by the digitizer represent the amplitude of the signal at discrete points in time. Each sample is stored as one element in a 4041 array.*

Array Processing

When data is processed, the 4041 manipulates only the numbers stored in a numeric array 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 a numeric array called WAVEFORM in the 4041. Now, 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 4041 BASIC statement is:

```
WAVEFORM=WAVEFORM+4
```

When this statement is executed, the 4041 adds four to the first element in the array (WAVEFORM) and stores the result in the first element. Then, it adds four to the second element, then the third, and so on until all the elements have been processed.

The 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. For example, when two arrays are added, the first element of one array is added to the first element of the other array and the result is stored in the first element of the result array. In the following statement, the first element of ARRAY1 is added to the first element of ARRAY2 and the result is stored in the first element of ARRAY3.

```
ARRAY3=ARRAY1+ARRAY2
```

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

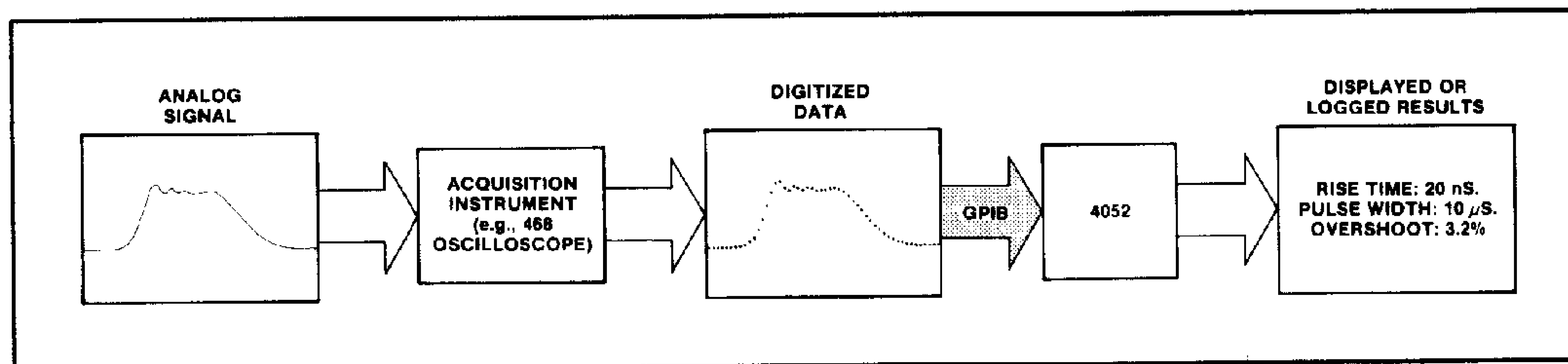


Fig. 6-1. *Automated measurement begins with the acquisition instrument. It sends digitized data to the 4041. The data may either be logged or processed and the results logged or displayed.*

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 4041 will remind you by printing an error message.

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

AVOID performing math operations on mixed data types, such as adding integer data to long floating-point data or multiplying long floating-point data by short floating-point data. Though these operations are valid and will not produce errors, they are slower than similar operations where both operands are of the same type. When mixed data types are used, the 4041 converts the lower precision operand to the same type as the higher precision operand before performing the operation.

If the destination variable type in a mixed-type operation is not explicitly declared, the result is returned in the higher precision type. If the type of the destination variable is explicitly set, the result is converted to that type before storage. As a result, if the type of the destination variable is lower precision than either of the operands, some precision is lost when the value is rounded to the shorter data type. Also, if an operation on two integers returns a non-integer number and the type of the destination is not specified, the result is returned in short floating-point format.

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 numeric errors or erroneous results.

DO keep in mind the largest and smallest values that can be stored in each type of numeric variable as well as the accuracy limits for each type. Table 6-1 lists the numeric limits and accuracy for each type of data. Normally, round-off errors in most calculations are not significant. However, in some lengthy calculations, the round-off error can accumulate to until it becomes significant. Always use a data type that ensures sufficient numeric accuracy for the application.

ROM Packs

The capabilities of standard 4041 BASIC can be extended by installing ROM (Read-Only Memory) packs in a carrier behind the grille at the bottom of the 4041. The ROM packs add graphics, plotting waveform processing, and other capabilities to the standard 4041. The waveform processing ROM, for example, adds routines for finding the minimum and maximum of an array, integrating and differentiating an array, and computing an FFT, just to mention a few. These routines are written in assembly language, so they execute much faster than equivalent BASIC programs. In addition, the ROM routines consume little or no user memory, so there's more room for data and BASIC programs.

Calling ROM pack routines. There are three ways to invoke ROM routines. They can be called with the RCALL (ROM Call) statement by specifying the routine name as a literal string in quotes (e.g.,

TABLE 6-1
4041 NUMERIC VARIABLE LIMITS

Type	Numeric limits	Accuracy limits
Integer	-32768 to +32767	Results rounded to integers
Short floating-point	±2.93874E-39 to ±3.40282E+38	Six decimal digits
Long floating-point	±5.562684646269E-309 to ±1.7976931348623E+308	14 decimal digits

"AMAX"). Or, the routine name can be stored in a string variable and the variable name can be used in the RCALL statement. In addition, the routine can be used directly in a statement without the RCALL keyword by using the literal name without quotes. The three options are illustrated below.

FORM 1—
Using the RCALL statement with a literal name:

```
Rcall "Routine name",parameter[,parameter...]
```

Example:

```
Rcall "ASK COLOR",number1,number2
```

FORM 2—
Using the RCALL statement with a string variable:

```
Name$="routine name"
Rcall name$,parameter[,parameter...]
```

Example:

```
Name$="ASK COLOR"
Rcall name$,number1,number2
```

FORM 3—
Calling the routine without RCALL:

```
ROUTINE_NAME parameter[,parameter...]
```

Example:

```
ASK_COLO number1,number2
```

There are several minor syntax differences when using the third form. Notice that if the routine name is longer than eight characters, it must be abbreviated to exactly eight characters. Also, if the routine name contains any imbedded spaces, an underscore must be substituted for the space. The first comma after the routine name is replaced with a space and the quotes around the routine name are omitted.

The number and type of parameters required in the call depend on the specific routine used. Some of the variables in the parameter list are used to pass input data to the routine while others return results from the routine. Some parameters are optional.

ROM Pack routine names do not interfere with variables of the same name. For example, you can use a variable named AMAX as well as call the AMAX routine from the 4041R03 ROM Pack in the same program. 4041 BASIC senses the difference between the two by the syntax of the statement in which the name is used. In addition, ROM pack routine names are printed in upper case in a listing. This makes it easier to distinguish ROM calls from program variables.

In a few cases, a conflict may exist between ROM calls executed in immediate mode and a variable name typed in immediate mode. In these cases, the ROM call names take precedence over equivalent variable names. For example, consider the following statements:

```
*100 page=1
*run
*page
```

If the 4041 does not have the 4041R01 Graphics ROM installed, the last immediate mode statement returns a value of 1.0 (the value of the numeric variable "page"). If the 4041R01 is installed, the last PAGE command is interpreted as a call to the PAGE routine in the 4041R01 ROM Pack, and the terminal screen will be cleared.

ROM routine names are not checked until run time. As a result, you can enter a program on a 4041 without the ROM packs, and store the program on tape. Then the program can be moved to a 4041 with the ROM packs or the ROM packs can be installed for execution.

Graphing Data

The old adage "a picture is worth a thousand words" may be a bit hackneyed but it is 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 the same relationship can often be recognized at a glance.

The 4041R01 Graphics ROM pack and the 4041R02 Plotting ROM pack provides the 4041 with a complete, powerful set of low-level graphics primitives as well as high-level plotting routines for a variety of graphics devices. The low-level graphics primitives allow you to define an area on the plotter or terminal screen where the graphics will be drawn and to scale user data units into the appropriate units for the graphics terminal or plotter. Graphics can be generated with MOVE and DRAW commands.

In addition, the Plotting ROM provides a set of high-level commands, such as YPLOT and XYPLOT, that draw complete graphs including axes, labels,

scaling information, and legends. The Plotting ROM translates high-level commands into a series of primitives executed by the Graphics ROM. As a result, the 4041R02 Plotting ROM requires the 4041R01 Graphics ROM for operation.

4041R01 Graphics ROM. The Graphics ROM pack gives the 4041 the capability to generate graphics on any device that uses Tektronix 4010, 4020 (except 4023), or 4662 compatible graphics commands. The ROM includes routines that set up the graphics device and the environment in which the graphics will be generated as well as graphic primitive routines that perform basic graphic operations. The routines included in the 4041R01 ROM pack are briefly described below.

GINIT—sets up the 4041 for communication with Tektronix graphic devices. Parameters in the CALL describe the type of graphic device and the LUN associated with it.

GDEVICE—sets up the 4041 to communicate with non-Tektronix graphic devices. Parameters in the call describe the characteristics of the device (e.g., minimum and maximum X and Y coordinates, etc.), and the LUN that is associated with it.

COLOR—sets a color (for color terminals) or gray index (for monochrome terminals) or chooses which pen will be used for drawing (for plotters).

LINESTYLE—selects the style of line to be drawn (e.g., solid line, dotted line, etc.).

TEXTSIZE—selects the maximum size of text characters.

VIEWPORT—selects the physical limits of the area in which the graphics will be drawn.

WINDOW—specifies the limits of the user's coordinatesto be graphed and how those coordinates will be mapped into the selected viewport.

ASK COLOR—returns the latest requested color/gray index or pen index and the current color/gray index or pen index. If the user asks for a color/gray index that is not supported by the device, the index requested is stored and returned by this call. The actual current index is also returned.

ASK LINESTYLE—returns the current linestyle index.

ASK TEXTSIZE—returns the latest requested and current character sizes.

ASK VIEWPORT—returns the current viewport parameters.

ASK WINDOW—returns the current window parameters.

DRAW—draws a line from the current position to a specified point.

GIN—returns the current position of the terminal cursor or plotter pen.

GTEXT—writes characters into the graphic area.

HARDCOPY—issues a copy command to an attached hard copy unit.

HOME—moves the cursor or plotter pen to the "home" position (one character's height below the upper-left corner of the graphic device).

MOVE—moves the cursor or plotter pen to a specified point without drawing a line.

PAGE—moves the cursor or pen to the "home" position. On terminals, the screen is erased; on plotters with paper advance, the paper is advanced.

POINTER—turns the graphic cursor on (if the graphic device is a terminal) and waits for the user to position the cursor or plotter pen on the screen or plotter surface. Then, the coordinates of the position are input and stored in the variables specified in the POINTER call.

RDRAW—draws a line from the current position to a new position whose coordinates are given relative to the current position.

RMOVE—moves the cursor or plotter pen to a new position whose coordinates are given relative to the current position. No line is drawn.

4041R02 Plotting ROM. The 4041R02 Plotting ROM provides high-level plotting commands to generate graphs and charts. Commands are included to generate X-T graphs (graph an array versus the array index) and X-Y graphs (one array versus another array). In addition, routines are included to control a variety of graphing parameters, such as the type of axes used, the labels for the graph, the symbols used to generate the graphs, and the range of the values to be graphed.

The 4041R02 requires that the 4041R01 Graphics ROM also be installed. The commands included in the 4041R02 ROM are briefly described below.

XYPLOT—draws an axis and plots data from one source array (X) against the other source array (Y). An option is provided for labeling the graph.

YPLOT—draws an axis, and plots data from the source array against an increasing or decreasing X value. The X value can increase or decrease linearly or logarithmically. An option is provided for labeling the graph.

XYADD—plots one source array (X) against the other source array (Y) on an existing XYPLOT. This allows multiple plots on a single axis. An option is provided for labeling the new curve.

YADD—plots a source array against an increasing or decreasing X value on an existing YPLOT. This allows multiple plots on one axis. An option is provided for labeling the new curve.

GRESET—restores all Plotting parameters to their default values. The TITLE, XTITLE, YTITLE, SUBTITLE, and 4041R01 Graphics ROM parameters are not affected.

LEGEND—specifies the location of the upper-left hand corner of the legend box.

ASK LEGEND—returns the current LEGEND parameters.

OFFSET—specifies the starting X value and X increment to use with the YPLOT and YADD routines.

ASK OFFSET—returns the current OFFSET parameters.

SYMBOL—specifies the symbol used to mark data points on the graph. The frequency at which data points are displayed is also set.

ASK SYMBOL—returns the current SYMBOL parameters.

XGRID—specifies whether tic marks should be extended from the X-axis to form a grid.

ASK XGRID—returns the current XGRID parameters.

XLABEL—specifies labels to use along the X axis with the next XYPLOT or YPLOT executed.

ASK XLABEL—returns the current XLABEL parameters.

XLOG—specifies whether the X axis should be a linear or a logarithmic scale.

ASK XLOG—returns the current XLOG parameter.

XRANGE—specifies the range of X values to plot.

ASK XRANGE—returns the current XRANGE parameters.

XTIC—specifies the starting value and increment of tic marks on the X axis.

ASK XTIC—returns the current XTIC parameters.

YGRID—specifies whether the tic marks should be extended from the Y axis to form a grid.

ASK YGRID—returns the current YGRID parameters.

YLABEL—specifies labels to use along the Y axis with the next YPLOT executed.

ASK YLABEL—returns the current YLABEL parameters.

YLOG—specifies whether the Y axis should be a linear or logarithmic scale.

ASK YLOG—returns the current YLOG parameters.

YRANGE—specifies the range of Y values to plot.

ASK YRANGE—returns the current YRANGE parameters.

YTIC—specifies the starting value and increment of tic marks on the Y axis.

ASK YTIC—returns the current YTIC parameters.

TITLE—writes the specified title above the graph.

XTITLE—writes the specified title below the X axis.

YTITLE—writes the specified title to the left of the Y axis. The title is written vertically from top to bottom.

SUBTITLE—writes the specified subtitle below the main title above the graph.

Signal Processing

Many GPIB system applications require processing operations considerably more complex than simple mathematical combinations of constants and waveforms. For example, a system might be required to measure the rise time of an acquired signal. This operation requires finding the top and base of the acquired pulse, computing the 10% and 90% points on the rise, and finding their time difference. This type of processing requires

several operations such as finding the maximum or minimum of an array or finding the point where an array crosses a specified value. Other operations, such as integration, differentiation, or FFT (fast Fourier transform) may also be required in some applications. Many of these operations are cumbersome and slow if implemented in BASIC.

The 4041R03 Signal Processing ROM Pack implements most common signal processing functions. These ROM routines are easier to use and much faster than equivalent BASIC programs. For example, Fig. 6-3 shows a BASIC routine that finds the maximum value of an array and its location. Next to the BASIC routine is a call to the 4041R03 ROM routine that performs the same function. For a 2048-point array, the BASIC routine takes about 20 seconds to execute, while the ROM routine takes about 0.72 seconds.

4041R03 Signal Processing ROM. The routines included in the Signal Processing ROM are listed and briefly described below. The routines and required arguments are described fully in the 4041R03 Signal Processing ROM manual.

AMAX—find the largest element of the source array and its location in the array.

AMIN—find the smallest element of the source array and its location in the array.

CONVL—convolve two input arrays, placing the results in the third (output) array. The input arrays are overwritten by intermediate results.

CORR—correlate two input arrays, placing the results in the third (output) array. The input arrays are overwritten by intermediate results.

CROSS—find the location of a specified crossing level within the input array.

DIF2—performs a two-point differentiation on the input array. The results are placed in the output array.

DIF3—performs a three-point differentiation on the input array. The results are placed in the output array.

FFT—computes a fast Fourier transform of an array. The original array is overwritten by the results.

IFT—computes the inverse Fourier transform of an array. The original array is overwritten by the results.

INLEAV—interleaves two input arrays, one containing real and one containing imaginary components, in the proper format for performing an IFT.

INTEGRAT—integrates the input array. The results are placed in the output array.

INTERP1—interpolates the value of a point that lies between two points in an array. The value of the point is computed using a windowed SIN(X)/X time-domain convolution.

MEANSTDV—computes the mean and standard deviation of the input array, returning the results in the two output variables.

```
100 Sub amax(array var maxval,maxloc)
110     Length$=ask$("VAR",array)
120     Posit=posn(length$,",",1,2)
130     Length=valc(length$,posit)
140     Maxval=array(1)
150     For i=2 to length
160        If maxval>array(i) then goto loop
170        maxval=array(i)
180        maxloc=i
190 Loop: next i
200     Return
210     End
```

a. *4041 BASIC program.*

```
AMAX array,maxval,maxloc
```

b. *4041R03 ROM Pack routine call.*

Fig. 6-3. *A 4041 BASIC program that finds the maximum value of an array and its location. The same function can be performed much faster with the AMAX routine in the 4041R03 Signal Processing ROM Pack.*

POLAR—converts an array of FFT data from rectangular form (real and imaginary components) to polar form (magnitude and phase components). The source and destination arrays are interleaved in the FFT format.

POLAR2—converts two arrays of unleaved FFT data (one array of real components and one array of imaginary components) into two arrays in polar form. One result array contains the magnitude components and the other contains the phase components.

RECTANG2—converts two arrays (one magnitude array and one phase array) to rectangular form. The result is one array of real components and one array of imaginary components.

TAPER—multiplies the input array by a cosine window of selected tapering weights. When the taper weight is set to 50%, the routine provides a Hanning window.

UNLEAV—sorts an array of interleaved FFT data (as returned by the FFT routine) into two arrays, one containing real and one containing imaginary components.

Sample Program

The program shown in Fig. 6-4 acquires a waveform from a 7612D Programmable Digitizer, scales the waveform data into volts, computes pulse parameters (rise time, fall time, and pulse width), and graphs the waveform. Several ROM routine calls are illustrated from the 4041R03 Signal Processing ROM as well as the 4041R02 Plotting ROM.

The program gets under way by opening a logical unit number (#3) for the digitizer and assigning this LUN to the variable DIGLUN. All I/O with the 7612D is performed through this LUN.

Line 120 gets the current number of records and length-of-records (number of samples/record)

```
100  Diglun=3
110  Open #diglun:"GPIB0(pri=3,sec=0):"
120  Input #diglun prompt "REC?":numrec,reclen
130  Integer waveform(numrec*reclen)
140  Dim wfmfloat(numrec*reclen)
150  Input #diglun using "5a,8e" prompt "VSL1?":volts$,voltsdiv
160  Input #diglun prompt "SBPT?":breakpt,samplint
170  Input #diglun prompt "ARM A;READ A" using "+8%,X":waveform
180  Wfmfloat=waveform*voltsdiv/32.0
190  AMIN wfmfloat,minval,minloc
200  AMAX wfmfloat,maxval,maxloc
210  Amplitud=maxval-minval
220  CROSS wfmfloat,(0.9*amplitud)+minval,time90
230  CROSS wfmfloat,(0.1*amplitud)+minval,time10
240  Risetime=(time90-time10)*samplint
250  CROSS wfmfloat,(0.9*amplitud)+minval,time90,2
260  CROSS wfmfloat,(0.1*amplitud)+minval,time10,2
270  Falltime=(time10-time90)*samplint
280  CROSS wfmfloat,(0.5*amplitud)+minval,time50
290  CROSS wfmfloat,(0.5*amplitud)+minval,time502,2
300  Pulswide=(time502-time50)*samplint
310  GINIT 100,4025,1
320  OFFSET 0,samplint
330  YPLOT wfmfloat
340  XTITLE "Seconds"
350  YTITLE "Volts"
360  Print using "fa,6j,fa":"Rise time = ";risetime;"s."
370  Print using "fa,6j,fa":"Fall time = ";falltime;"s."
380  Print using "fa,6j,fa":"Pulse width = ";pulswide;"s."
390  HARDCOPY
```

Fig. 6-4. *This program acquires a pulse using a 7612D Programmable Digitizer, computes the pulse parameters (rise time, fall time, and pulse width), and graphs the pulse on a graphics terminal. Several 4041 ROM calls are illustrated.*

settings from the digitizer using the REC? query. The program assumes that all records will be included in the processing and graphics. Two arrays, one integer and one floating-point, are dimensioned to the size required to hold all records. This size is computed by multiplying the number of records times the size of each record.

Next, line 150 gets the vertical scale factor (volts/div setting) for channel 1 using the VSL1? query. The INPUT statement reads a string variable and a numeric variable (VOLTS$ and VOLTSDIV). The VOLTS$ string receives the header returned from the query (VLS1) and the VOLTSDIV variable receives the volts/div setting in scientific notation. Normally, the VOLTS$ string would not be necessary; a numeric variable would be all that is required. The header information would be ignored and only the numeric value would be stored. However, since the header contains a number (1), it must be read. Otherwise, the numeric variable will get the value 1 instead of the volts/div setting.

The USING clause specifies that five characters of string data will be read into VOLTS$ ("5a"), followed by a numeric value in scientific notation ("8e").

Line 160 gets the sampling interval using the breakpoint query (SBPT?). This program assumes that only one sampling interval is used. Thus the SBPT? query would return a breakpoint at sample zero and the sampling interval for the entire record. The first numeric variable in line 160 gets the breakpoint location (0) and the second variable gets the sampling interval value.

At this point, the program has the vertical (volts/div) and horizontal (sampling interval) scale factors. Line 170 starts the acquisition. The ARM and READ commands are sent together, so as soon as the acquisition is complete, the 7612D begins sending data. The USING clause in line 170 tells the 4041 to expect 8-bit binary data sent in the block binary format and to ignore the termination semicolon the 7612D sends after a binary block.

Once the waveform data has been read into the integer array (WAVEFORM), line 180 converts the data to volts by multiplying each array value by the volts/div setting and dividing the results by the number of digitizer levels per division (32). The results of this calculation are stored in a floating-point array for additional accuracy.

Lines 190 through 300 compute pulse parameters for the input waveform. The waveform is assumed to be a relatively clean pulse that is completely contained within the acquisition window (see Fig. 6-5). First, line 190 and 200 find the maximum and minimum value of the pulse using the 4041R03 Signal Processing ROM AMAX and AMIN routines. Line 210 computes the pulse amplitude by subtracting the minimum value from the maximum value. Then, the 10% and 90% points on the leading edge of the pulse are found using the CROSS routine. The rise time is the time at the 90% point minus the time at the 10% point. The same technique is used to find the fall time. The pulse width is the time between the two 50% points on the waveform.

Finally, the program plots the waveform on a 4025 Graphics Terminal attached to the COMM0 port. Line 310 initializes the plotting ROM and the plot values. It also tells the ROM what type of graphic device is being used. The ROM supports all Tektronix graphic terminals and plotters.

The OFFSET statement in line 320 sets the value of the X-axis labels. By default, the waveform data would be plotted against the array index on the X axis. OFFSET changes the X axis scaling to represent seconds just as it would on an oscilloscope display. The first value specifies the beginning X value. If pre- or post-trigger is used, the X offset can be set to reflect this value. The second value in the OFFSET statement indicates the value of each X index. In this case, each X value is equivalent to one sampling interval.

Finally, line 330 draws the axes, labels them and plots the data on the axes. The XTITLE and YTITLE routines write titles for the X and Y axis, respectively. The pulse parameter results are printed below the graph in engineering notation. Figure 6-5 shows typical output from the program.

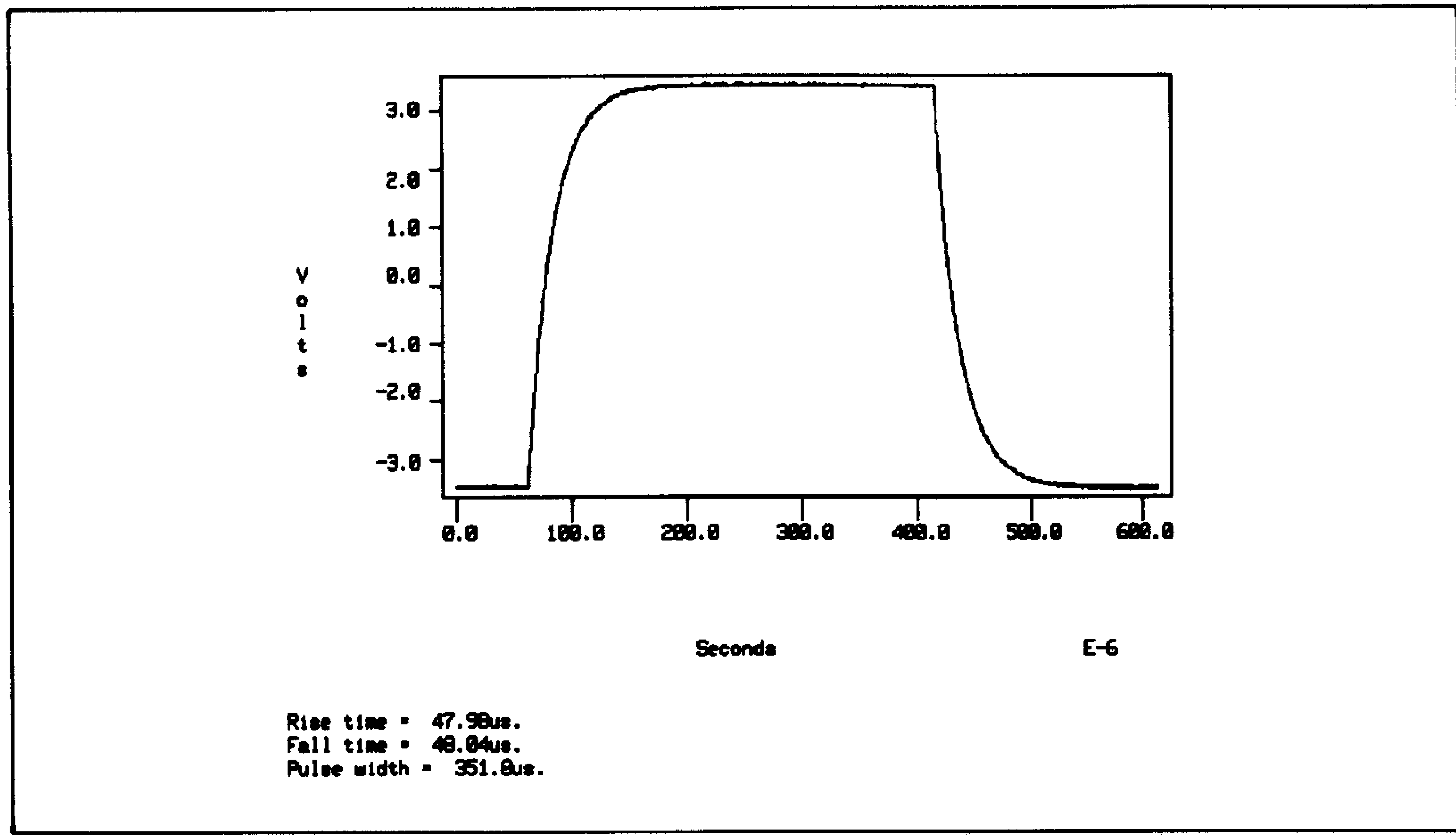


Fig. 6-5. *Typical output from the program in Fig. 6-4.*

Section 7—Estimating 4041 GPIB System Performance

"How fast will it go?" That question plagues many people trying to specify or assemble GPIB systems. The answer is often critical to the design and implementation of a system. Yet, it is often a very difficult question to answer. The complete performance picture is composed of a multitude of parts, and many of the parts are difficult to estimate.

The key to understanding GPIB system performance is in breaking the performance picture into its component parts. This section introduces you to the major components of GPIB system performance. Each component is briefly described. Then, some techniques for estimating the contribution of each part to your system's performance are discussed.

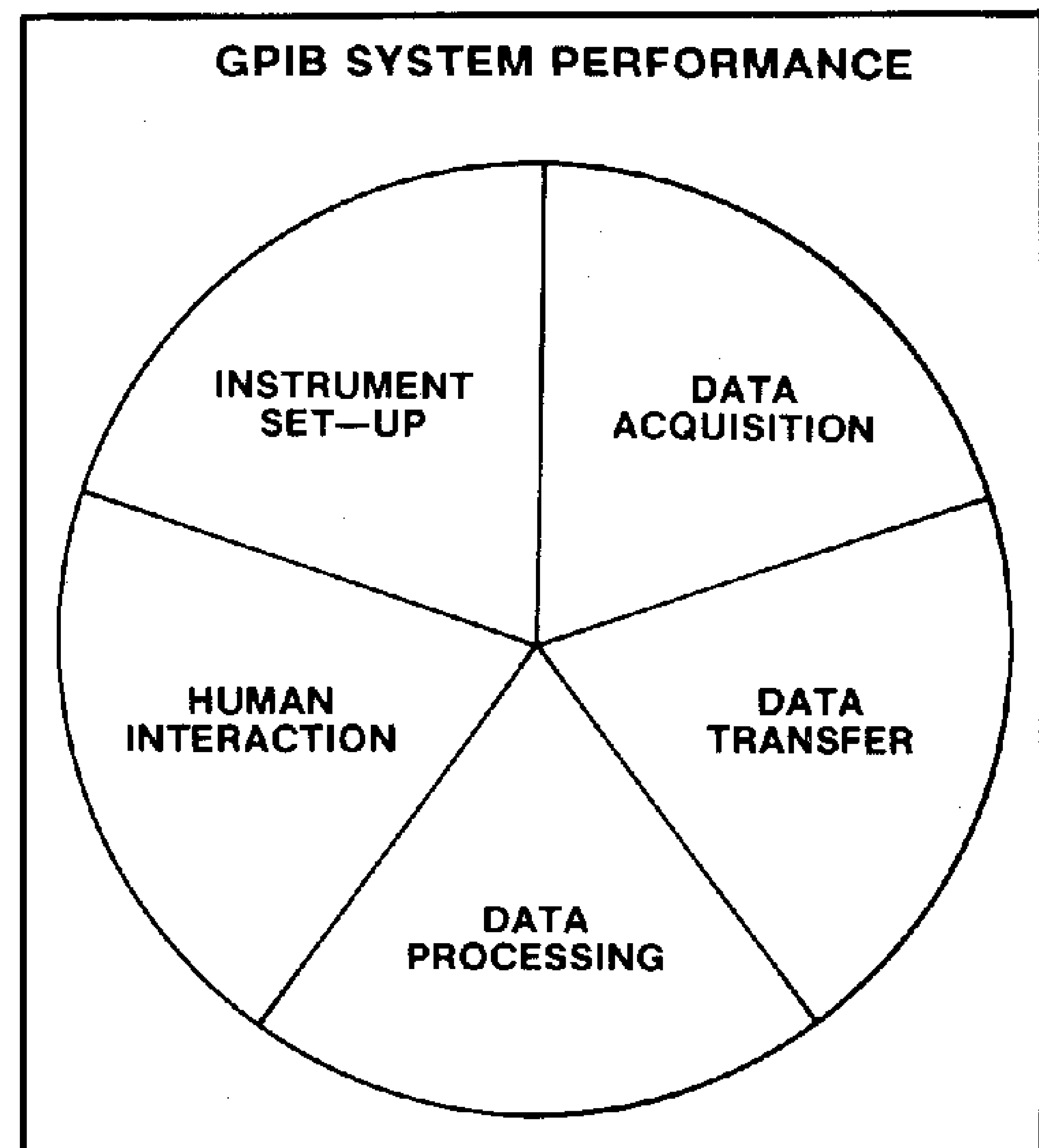


Fig. 7-1. *The GPIB performance picture is composed of five major parts. The relative amount of time spent on each component varies with different systems.*

GPIB System Performance Factors

Five major components compose the overall GPIB system performance picture (Fig. 7-1). The sum of these components is the total time required for the system to execute its intended function. Some systems may not have all these components, and the amount of time required for each component varies between different systems and tasks. For example, a data logging system probably spends very little time setting up instruments, and it usually doesn't require any operator interaction. Most of its time is spent acquiring and transferring data. In contrast, a production test system may spend less time acquiring data and spend more time processing data and interacting with the operator.

Instrument set-up time. One of the first tasks for a system controller is controlling instrument settings. For example, the controller might have to change the output frequency of a function generator several times during a test, or it may have to change the trigger level of a digitizer. Any time programmable settings are changed, the instrument requires some time to interpret the commands, make the settings changes, and some additional time for the inputs or outputs to settle to the new setting. The total time required for the instrument to interpret and execute a setting command and for the new settings to stabilize is called set-up time.

Set-up time can be divided into two parts—the time required to decode and execute a setting command, and the time required for the new settings to stabilize. For example, a programmable power supply may require a few milliseconds to decode and execute a command such as "VPOS 10" that sets the power supply output to 10 volts.

The time to decode and execute a setting command is usually small, but more time may be required if a command initiates a complex or lengthy instrument operation. The TEKTRONIX 7612D Programmable Digitizer, for instance, requires only about 300 microseconds to execute an ARM command if no settings have been changed. However, if many settings have been changed, the ARM command can take 50 milliseconds or more to execute because the instrument must check the new settings and load them into the hardware.

The second part of set-up time is settling time (sometimes called Step Response Time). This is the time required for the instrument to settle to a specified accuracy after an input or setting change. In the case of the power supply, settling time is the time required for the supply to settle to the new output value within the specified accuracy. For a measurement instrument, such as a DMM, it is the time required for the instrument to return a reading within the specified accuracy after a range or input change.

In auto-ranging instruments, such as DMMs, some additional time may be required for automatic range changes. For each range change, the instrument usually takes a reading, tests for under- or over-range conditions, steps the range up or down, and takes another reading. If the reading is within the new range, the process stops. Otherwise, another range change may be made. For each automatic range change, at least one reading must be captured. In some cases, two or more readings may be required for each range change.

Data acquisition time. The second major component of GPIB system performance is the time required to acquire data. Two factors impact the data acquisition time: trigger delay and digitizing time.

Trigger delay is the amount of time between when the instrument is ready to begin acquiring data and when the trigger occurs (Fig. 7-2). The trigger for a logic analyzer may be a specified pattern of bits on its inputs or, for a digitizer, the trigger might be an input voltage level. Triggers may also be generated by an external source or a command from the GPIB, such as the group execute trigger (GET) message.

Trigger delay may also include the time some instruments require for capturing pre-trigger samples. For example, the 7612D Programmable Digitizer acquires one full record of data in pre-trigger mode before it can be triggered. If a trigger occurs while the pre-trigger acquisition is in progress, the trigger is ignored.

Once a valid trigger occurs, the instrument begins acquiring data. The time required to actually capture and store input data is called digitizing time. In some instruments, such as DMMs, a single sample is captured, so the digitizing time is simply the conversion time of the A/D converter. The DM 5010 Programmable Digital Multimeter, for example, takes about 300 milliseconds to capture one reading in normal conversion rate and DC volts mode.

Digitizing time in waveform digitizers involves capturing many samples either in rapid succession or spread over several successive repetitions of the input waveform.

In addition, some acquisition instruments (digitizers, DMMs, etc.) perform on-board signal averaging. When signal averaging is performed, several complete acquisitions are performed and the results are averaged. Each acquisition usually requires a separate trigger (or several triggers for equivalent-time digitizing), so the complete acquisition time is multiplied by the number of averages performed. Thus, if 32 averages are performed, the acquisition time (trigger delay plus digitizing time) is about 32 times the single acquisition time.

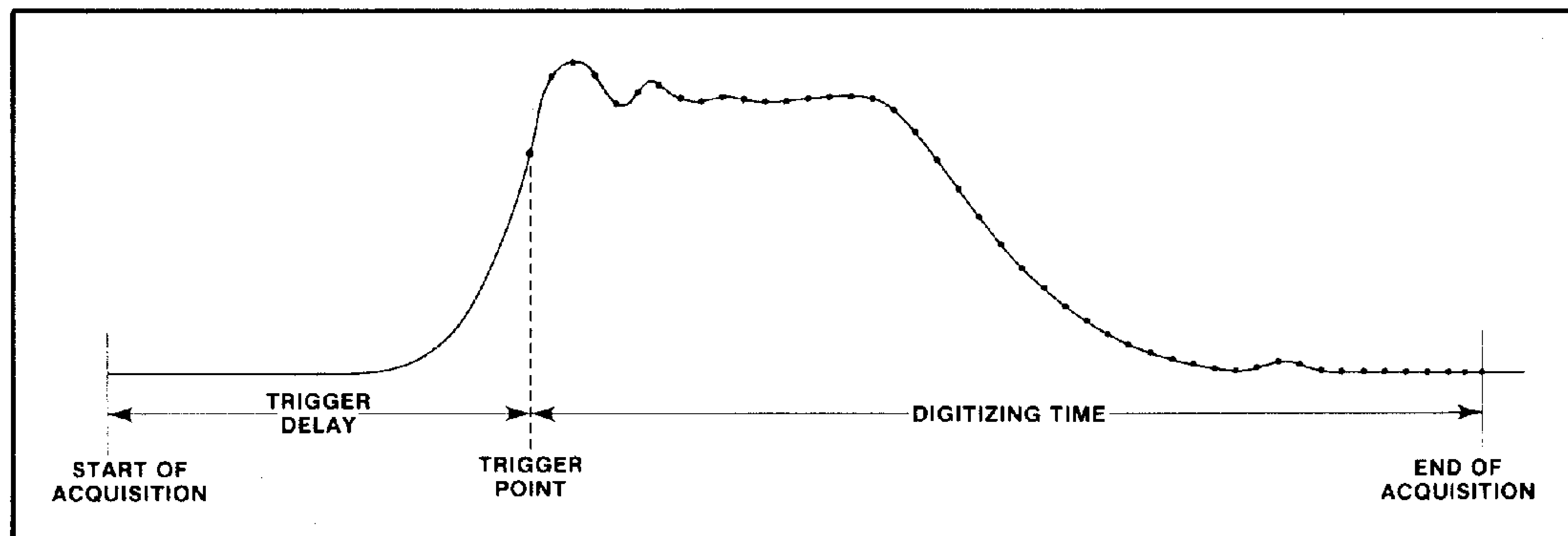


Fig. 7-2. *Data acquisition is composed of two parts. The first part, trigger delay, is the time between when the instrument is ready to begin acquisition and when the trigger occurs. The second part, digitizing time, is the time required to actually capture the data.*

Data transfer time. A certain amount of time is required to transfer data over the GPIB. This data may include interface messages (involving all devices on the bus) or device-dependent commands or data (involving only the addressed talker and listeners). The time required to transfer data over the GPIB is determined by two factors: the number of bytes that are transferred and the time to transfer each byte.

The number of bytes transferred depends on the message being transferred. The rate at which bytes are transferred depends on the devices involved in the transfer. The slowest device involved in a transfer limits the rate of the transfer. As a result, the maximum transfer rates of some devices may not be realized.

Figure 7-3 shows a typical GPIB system with a controller, a digitizer, and a magnetic tape drive. The maximum GPIB data transfer rate for each device is shown. During interface message transfers (addressing, unaddressing, bus control, etc.), all devices are involved, so the transfer is limited to a maximum rate of 800 bytes/second (the rate of the tape drive).

When device-dependent data is being transferred between the controller and the digitizer, the transfer can proceed at up to 7500 bytes/second (the rate of the controller). During device-dependent transfers between the controller and tape drive, the maximum rate is limited to 800 bytes/second (the rate of the tape drive). Notice that unaddressed devices do not affect the device-dependent data transfer rate.

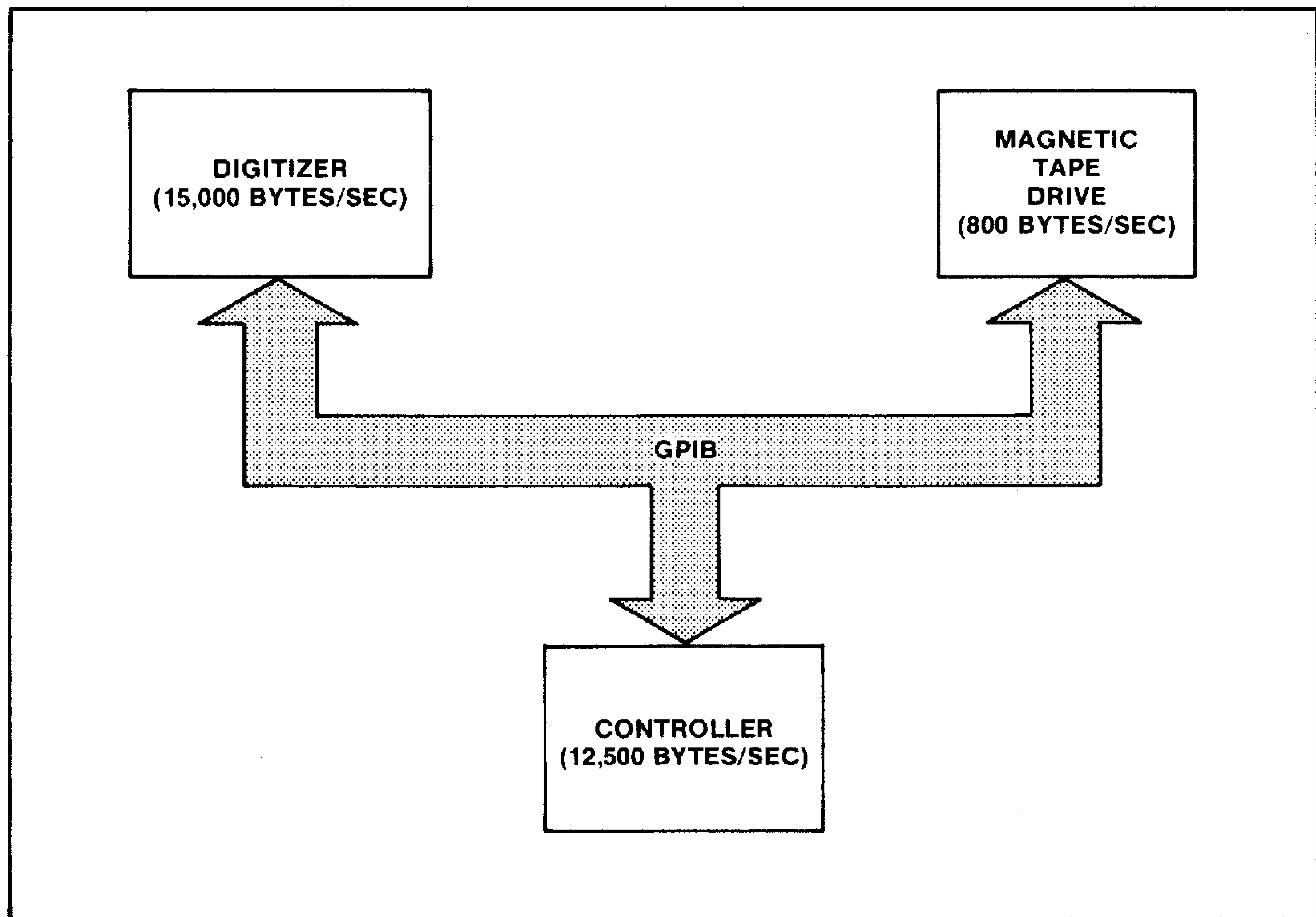


Fig. 7-3. *A typical GPIB system showing the maximum data transfer rates for each device in the system. All transfers are limited to the speed of the slowest device involved.*

This asynchronous nature of the GPIB allows a variety of instruments with different speeds to work together. But, it also means that slow devices involved in a transfer limit the rate of the transfer. Thus, the maximum transfer rate for a GPIB instrument listed in the specifications may not be the actual data transfer rate if it communicates with a slower device.

Data processing time. The fourth major factor in system performance is the time required to manipulate acquired data to arrive at the desired results. In modern GPIB systems, this task is often shared between the system controller and intelligent instruments. For example, the TEKTRONIX 7854 Programmable Oscilloscope can store and execute its own program, sending partially processed data or measurement results to the controller.

The time required to process data depends entirely on the complexity of the task, the speed of the computer, and the number and complexity of other tasks the computer is assigned. In addition, programs that use assembly language or ROM routines are generally faster than routines written in BASIC.

Human interaction time. The fastest automated system can be slowed to a snail's pace if it requires excessive interaction with human operators. Operators may be required to enter test parameters, make adjustments to non-programmable instruments attach leads to a device-under-test, or set up tests. This interaction time can easily become the largest part of a system's total operating time.

Estimating Performance Factors

Armed with a basic understanding of the factors that affect system performance, you can begin to make some reasonable estimates of how much time your system spends in each of these areas. A few suggestions are provided here for developing performance estimates in each area. Suggestions for improving performance in each area are provided in Section 8.

Estimating instrument set-up time. The time required for instrument set-up is often difficult to estimate since most of the operations involved are internal to the instruments. Fortunately, in most systems the set-up time is a relatively small part of the total system performance. Only when a large number of settings changes are made does the set-up time normally become significant. However, there are a few things you can do to estimate the time involved.

First, check the instrument manuals. The set-up time, or at least part of it, may be specified. For example, the settling time for the DM 5010 Programmable Digital Multimeter is specified in the manual (it's called "Settling Time"). This specification is reproduced in Table 7-1. The settling times listed in this table do not include command decoding and execution time—it's just the maximum time from an input change to when the next valid reading will be available.

The command decoding and execution time depends a great deal on the context of the command. A command included as part of a larger string of commands takes longer to decode and execute than a single command. In addition, some instruments allow you to abbreviate full command names to a few unique characters. Normally, the abbreviated commands are processed by the instrument faster than the full command.

The command decoding and execution time usually isn't specified since it depends so much on the context and format of the command. However, if

TABLE 7-1
DM 5010 STEP RESPONSE TIMES

Conversion Mode	Measurement Mode		
	AC Volts	DC Volts	Ohms
Normal Rate Run Mode	1.2 sec	0.53 sec	1.24 sec
Fast Rate Run Mode	1.2 sec	0.08 sec	0.33 sec
Normal Rate Triggered Mode	1.2 sec	0.33 sec	0.73 sec
Fast Rate Triggered Mode	1.2 sec	0.06 sec	0.19 sec

the total set-up time is an important issue in your system, it can be estimated by measuring the time from when the message terminator is received to when the instrument's input or output settles to the new value. For example, with the PS 5010 Programmable Power Supply, the time from when EOI is asserted at the end of a command to when the supply output settles to the new value can be measured with a counter/timer or oscilloscope. Table 7-2 shows the results of such a measurement when the PS 5010 positive supply output is being changed from zero volts to one volt with no output load.

TABLE 7-2
PS 5010 SET-UP TIME MEASUREMENTS

Command Sent	Set-up Time
"VP 1"	17.1 ms
"VPOS 1"	18.0 ms
"VNEG 1;VPOS 1"*	26.0 ms

* *The "VNEG 1" command doesn't affect the positive supply, it simply illustrates what happens to set-up time when a command is included in a longer string of commands.*

In any set-up measurements like the ones suggested previously, it is important to note the test conditions. A minor change in test conditions—such as the load on a power supply or the length and context of the command sent to the instrument—can have a significant effect on the test results.

Estimating data acquisition time. Remember from the previous discussion that there are two parts to data acquisition—trigger delay and digitizing time. The trigger delay is the amount of time from when the instrument is ready to begin acquiring data to when the trigger actually occurs. The digitizing time is the amount of time required to actually sample and store the data in the digitizer memory.

The trigger for an acquisition usually comes from one of three sources:

- the signal itself
- an external event
- the controller

In each case, the key to estimating trigger delay is knowing the characteristics of your trigger. For triggers taken directly from the input signal, the **maximum** trigger delay is usually the reciprocal of the signal's period (assuming that it is periodic). If for example, a signal has a repetition rate of 10 Hertz, the maximum trigger delay would be 1/10 Hertz or 100 milliseconds.

For single-shot or transient events, or when an external event or the controller triggers the acquisition, the trigger delay is the amount of time from when the digitizer is ready to begin acquiring data to when the trigger event occurs.

The amount of time required to sample the data (digitizing time) depends on the input signal and the type of acquisition instrument. For a DMM, the digitizing time is simply the reciprocal of the conversion rate for the DMM. If the DMM is specified to take 30 readings per second, for example, the digitizing time is about 33 milliseconds.

For waveform digitizers, the question of digitizing time is a little more complex. There are two basic types of digitizers and the approach to estimating digitizing time is a different for each type.

The first type is called a sequential digitizer. As its name implies, this type captures all its samples in sequence on a single pass. The input signal does not have to be repetitive because the complete waveform is acquired on a single pass (Fig. 7-4). The digitizing time for this type of digitizer is the number of sample intervals (the number of samples minus one) times the sampling interval. If a sequential digitizer captures 2048 samples at 100 nanoseconds per sample, the digitizing time is:

$$(2048-1) \times 10 \text{ ns} = 20.47 \text{ microseconds}$$

Equivalent-time digitizers capture their samples in a little different way. They capture one or more samples on each repetition of the input signal. Samples from each repetition or sweep are stored in the digitizer's memory. On each repetition of the waveform, more samples are captured and stored until the complete waveform is sampled. This method allows the digitizer to sample a signal at a slower rate than a sequential digitizer, but it requires that the input signal be repetitive. Figure 7-4 illustrates the difference between a sequential digitizer and an equivalent-time digitizer.

The digitizing time for an equivalent-time digitizer depends on how many repetitions are required to capture the signal and the repetition rate of the

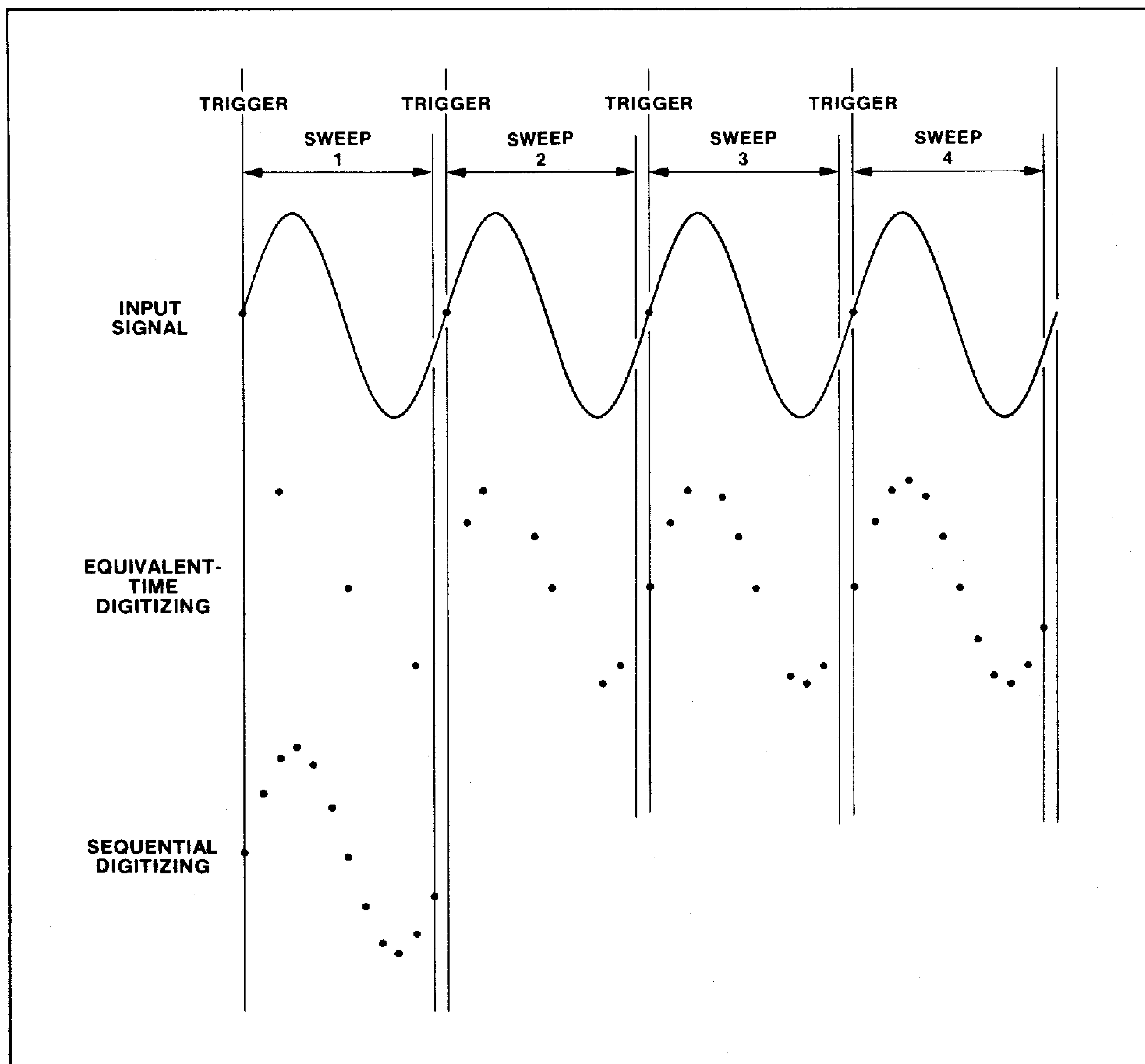


Fig. 7-4. *Equivalent-time digitizers sample a repetitive signal on several successive sweeps, gradually collecting enough samples to fully define the waveform. Sequential digitizers, on the other hand, acquire all the samples in a single pass. Equivalent-time digitizers typically have a much higher bandwidth potential than sequential digitizers because of the technique used.*

signal. Some equivalent-time digitizers capture only one sample per sweep regardless of the sweep speed, so the number of sweeps required is equal to the number of samples that are taken. Most, however, capture several samples per sweep. The exact number depends on the sweep rate. The higher the sweep rate, the fewer samples are captured per sweep and the more sweeps are required to complete the digitizing.

Each repetition or sweep also requires a separate trigger, so the trigger delay is multiplied by the number of repetitions.

Table 7-3 shows the average number of triggers required by the TEKTRONIX 7D20 Programmable Digitizer to capture a complete waveform in equivalent-time digitizing mode. (The 7D20 uses sequential sampling at TIME/DIV settings slower

TABLE 7-3
7D20 EQUIVALENT-TIME
DIGITIZING MODE TRIGGER REQUIREMENTS

TIME/DIV setting	Number of acquired points/trigger	MINIMUM triggers/waveform	AVERAGE triggers/waveform
1 μs	204/205	5	11
500 ns	102/103	10	29
200 ns	40/41	25	95
100 ns	20/21	50	224
50 ns	10/11	100	516

than 1 microsecond/div.) Notice that the number of triggers required increases and the number of points acquired per trigger decreases with faster TIME/DIV settings.

The table shows the minimum and average number of triggers—not the exact or maximum number of triggers. However, the average value can be used to develop a reasonable estimate of the digitizing time. The average acquisition time can be computed by adding the trigger delay and the time to complete one repetition (TIME/DIV setting times 10 divisions). This sum is multiplied by the number of repetitions (triggers) required to complete the acquisition. Thus, the average acquisition time is:

Acq time = (trig delay + (TIME/DIV * 10 div)) * no. of trig

If, for example, a pulse with a repetition rate of 10 kilohertz is being digitized on the 200 nanoseconds/div setting, the acquisition time can be estimated as shown below.

Trigger delay = 1/10 KHz.
= 100 μs.

Dig. time = 200 ns./div * 10 div.
= 2 μs.

Acq. time = (Trig. delay + Dig. time) * no. of triggers
= (100 μs. + 2 μs.) * 25
= 2.5 ms.

Notice that trigger delay in an equivalent-time digitizer can be a large part of the total acquisition time since each repetition requires a separate trigger. The lower the repetition rate of the input signal, the longer the acquisition takes. In the above example, the acquisition time jumps to 25 milliseconds if the repetition rate of the signal drops to 1 kilohertz.

Estimating data transfer time. Developing an estimate of data transfer time begins with a solid understanding of how the 4041 handles GPIB I/O. When a 4041 GPIB I/O statement, such as PRINT or INPUT, is executed, the execution of the I/O statement can be broken into five parts. Some I/O statements don't include all these parts, but this basic framework can be applied to all 4041 GPIB I/O. The five events are illustrated in Fig. 7-5.

Statement Overhead—when an I/O statement is executed, the 4041 first examines the statement for content and syntax. User-defined functions and other expressions used in the I/O list are evaluated first. Then, any clauses included in the statement (e.g., BUFFER, USING, etc.) are evaluated.

Next, the 4041 establishes the path for the I/O operation. If a LUN is not included in the statement, a link is set up to direct the I/O to the console device. If a LUN is included but it is not open, the LUN is implicitly opened. If the LUN is open, a link is set up for that LUN. If a stream specification is included instead of a LUN, the stream spec is parsed.

When the I/O path is set-up, an I/O buffer must be allocated. If a BUFFER clause is included in the statement, the specified string variable is used as an I/O buffer. Otherwise, a buffer is allocated from free memory.

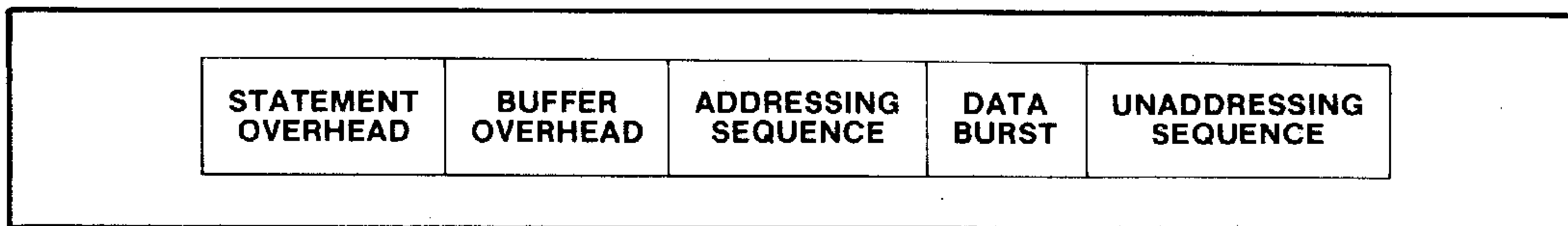


Fig. 7-5. *GPIB data transfers are composed of five basic parts, though the parts are not always executed in the order shown.*

The statement overhead is the first processing performed when an I/O statement is executed. The first activity on the GPIB marks the end of the statement overhead. The minimum statement overhead takes about 5 milliseconds. For PRINT and INPUT statements, the basic overhead time is about 8 milliseconds. Adding clauses to the statement increases the overhead time. Table 7-4 shows some typical statements and their overhead times.

TABLE 7-4
STATEMENT OVERHEAD
FOR TYPICAL I/O STATEMENTS

Statement	Overhead
PUTMEM BUFFER A$:	5.0 ms.
PUTMEM BUFFER ASING "FA":	6.6 ms.
PRINT ;	8.0 ms.
PRINT USING "1A":;	10.0 ms.
PRINT USING "1A" BUFFER B$:;	10.2 ms.

A simple way to estimate statement overhead for a particular statement is to measure the amount of time (using the Ask("TIME") function) required to execute the statement without an I/O list and no stream spec or LUN. This eliminates the addressing sequence, buffer overhead, and data transfer time. For example, consider the following statement:

```
PRINT #1 USING "+8%":ARRAY
```

A reasonable estimate of the statement overhead can be made by measuring the time (using the ASK("TIME") function) required to execute the same statement without its I/O list or LUN:

```
PRINT USING "+8%":;
```

Buffer Overhead—all 4041 I/O passes through a special area of memory called the I/O buffer. This buffer is the "staging area" for I/O operations. On output, data is converted from the internal storage format to the required format for output. On input, data received from the I/O device is stored temporarily in the I/O buffer. Then, the received data is converted to internal storage format and stored in the specified variables.

The time required to fill the I/O buffer with data for output or empty it after input is called buffer overhead. On output statements, such as PRINT, buffer overhead occurs directly after the statement overhead. If more data is transferred than will fit in the I/O buffer, the transfer occurs in blocks the size of the I/O buffer. After each block is transmitted (except the last block), the buffer overhead occurs again (Fig. 7-6a).

On INPUT, the first buffer overhead period occurs after the first block of data is transferred. If the data transfer contains more data than will fit in the I/O buffer, the buffer overhead occurs after each block is transmitted, including the last block (Fig. 7-6b).

The time required for buffer overhead depends on the amount of data in the I/O buffer and the type of data. For string data, the overhead time depends on the number of individual strings and the length of the strings. The first string variable or constant takes a minimum of about 700 microseconds to process. Each additional string takes a minimum of about 1.4 milliseconds. Long strings take a little longer. Each character takes about 4 microseconds, so for a string of 100 characters, the overhead time is about 1.8 milliseconds (1.4 milliseconds + 100 * 4 microseconds).

Table 7-5 shows a few typical PUTMEM statements and their buffer overhead times. The statement overhead (5 milliseconds) has been subtracted from the times to show only the buffer overhead.

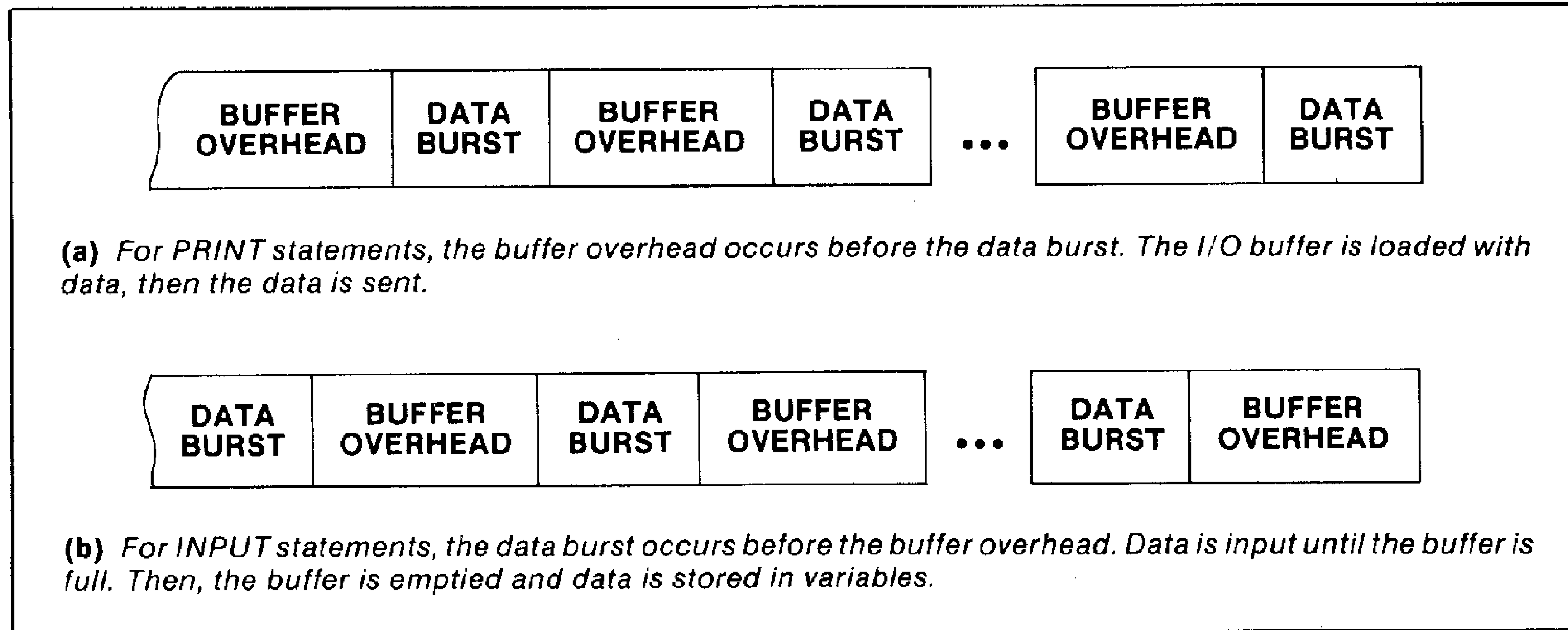


Fig. 7-6. *The buffer overhead and data burst occurs in opposite order for PRINT and INPUT statements. Data is transferred in bursts the size of the I/O buffer. If a large enough buffer is used with the BUFFER clause, the data is transferred in a single burst.*

TABLE 7-5
BUFFER OVERHEAD TIMES FOR STRINGS

Statement	Buffer Overhead
PUTMEM BUFFER A$:"A"	0.7 ms.
PUTMEM BUFFER A$:"<100-char. string>"	1.1 ms.
PUTMEM BUFFER A$:"A","B"	2.1 ms.
PUTMEM BUFFER A$:"A","B","C"	3.5 ms.
PUTMEM BUFFER A$:B$,C$,D$	3.5 ms.

Buffer overhead time for numeric variables can vary a great deal depending on the type of variables or constants, the size of the numeric values, and the output format. Numeric data can be output in variety of formats including various ASCII formats (scientific notation, fixed or floating decimal, etc.) as well as binary block.

In the simplest case—standard ASCII output (without a USING clause)—each integer variable or constant takes between 3.5 milliseconds and 8.8 milliseconds of overhead. Each short floating-point variable takes between 3.7 milliseconds and 15 milliseconds. Each long floating-point variable or constant takes between 4 milliseconds and 20 milliseconds, depending on the size of the values.

Numeric arrays are processed in the same manner as scalar variables, except that arrays are a little faster than an equivalent number of scalar variables. For example, a short floating-point array is processed at between 3.0 and 4.1 milliseconds per element compared to between 3.5 and 15 milliseconds per scalar variable.

Table 7-6 summarizes the overhead times for the various types of numeric data output in standard ASCII.

TABLE 7-6
BUFFER OVERHEAD
FOR NUMERIC DATA OUTPUT
IN ASCII

Data Type	Overhead Time
Integer scalars	3.5 - 8.8 ms
Short floating-point scalars	3.7 - 15.0 ms
Long floating-point scalars	4.0 - 20.0 ms
Integer arrays	3.0 - 4.1 ms/element
Short floating-point arrays	3.1 - 4.9 ms/element
Long floating-point arrays	3.2 - 10.8 ms/element

When the USING clause is used to control output data formatting, buffer overhead times depend on the output format chosen. For example, binary block data output requires considerably less overhead time since the conversion to output format is simpler and takes fewer bytes. Table 7-7 shows the approximate buffer overhead time per array element for the 4041's two binary block data formats. The total overhead time for an array is the value shown in the table multiplied by the number of array elements transmitted.

TABLE 7-7
BUFFER OVERHEAD
FOR NUMERIC DATA OUTPUT
IN BINARY

Data Format	Overhead Time
2-byte binary blocks (+16%)	55 μs/element
2-byte end binary block (+16@)	55 μs/element
1-byte binary block (+8%)	35 μs/element
1-byte end binary block (+8@)	35 μs/element

The buffer overhead for a particular statement can be estimated by measuring the time to execute an equivalent GETMEM or PUTMEM statement. For a PRINT statement, an equivalent PUTMEM statement performs the same buffer overhead process without doing any I/O. As a result, the time to execute the PUTMEM statement, minus the statement overhead time is approximately equal to the buffer overhead time. For example, consider the following PRINT statement:

```
PRINT #1 BUFFER A$ USING "+8%":ARRAY
```

The buffer overhead time for this statement can be estimated by measuring the execution time for the following statement:

```
PUTMEM BUFFER A$ USING "+8%":ARRAY
```

Remember to measure the statement overhead (by executing the PRINT statement with no I/O list or LUN) and subtract this value from the buffer overhead.

The same technique can be applied to INPUT statements, using the GETMEM statement as a measure of buffer overhead time. If the PRINT or INPUT statements do not use the BUFFER clause, the default I/O buffer is used, so you should dimension the buffer string in the PUTMEM or GETMEM statement to the same size as the default I/O buffer (514 bytes).

Addressing Sequence—the first activity on the bus is the addressing sequence. During this period, the 4041 issues the necessary talk and listen addresses to prepare for the data transfer. For INPUT statements, the 4041 sends the external device's talk address and its own listen address (MA+32). For PRINT statements, the 4041 sends the external device's listen address and its own talk address (MA+64).

All devices on the bus must handshake the address bytes, so the slowest device on the bus limits the transfer rate. The timing values listed here assume that the 4041 is sending bytes at its maximum rate (no devices are limiting the transfer speed). In addition, these timing values apply only to the addressing sequence generated automatically by PRINT and INPUT statements. For addresses sent with WBYTE, see **WBYTE and RBYTE Timing**.

When no secondary address is sent, the addressing sequence takes about 1.7 milliseconds. When a secondary address is included, the addressing sequence takes about 1.8 milliseconds.

INPUT statements using the PROMPT clause generate two addressing sequences—one before sending the prompt and one before accepting the response. These addressing sequences are similar to normal PRINT and INPUT addressing sequences, except that an UNListen command precedes each addressing sequence. Even with the additional byte, both sequences take about the same amount of time as a normal addressing sequence (about 1.7 milliseconds without a secondary address and about 1.8 milliseconds with a secondary address).

The addressing period is not affected by use of FAST, DMA, or PROCEED modes.

Data Burst—the data burst is the time when device-dependent data is actually being transferred between the 4041 and one or more external devices. Again, the transfer rate for this data is limited to the rate of the slowest device involved in the transfer. The values listed here are maximum values assuming that all other devices involved in the transfer are faster than the 4041, and therefore do not limit the transfer rate.

TABLE 7-8
4041 GPIB DATA BURST RATE

Transfer Mode	Std. GPIB	Option 1 GPIB
Normal mode input	5.0 K bytes	3.0 K bytes
Fast mode input	16.5 K bytes	12.5 K bytes
DMA mode input	*	600.0 K bytes
Normal mode output	5.0 K bytes	3.0 K bytes
Fast mode output	19.5 K bytes	18.5 K bytes
DMA mode output	*	240.0 K bytes

** DMA mode can only be used on the Option 1 GPIB port (GPIB1).*

Maximum data burst rates are different for each of the three I/O modes (NORmal, FASt, and DMA). Table 7-8 lists the maximum burst rates for each of the three modes.

The data burst rates indicate the number of bytes that can be transferred per second. The total data burst time is the number of bytes transferred divided by the data burst rate (in bytes/second). For example, if a PRINT statement sends 150 bytes on the bus in FASt mode, the **minimum** data burst time is:

Min. Burst Time = 150 bytes/19.5 K bytes per second
Min. Burst Time = 7.7 milliseconds

An accurate estimate of data burst time requires that you know how many bytes are being transferred in a particular message. For simple ASCII command messages, the number of bytes is simply the number of characters in the command. Don't forget to include spaces and formatting characters automatically added by the 4041 on output (EOH, EOA, EOU, EOQ, and EOM).

When ASCII waveforms or other numeric arrays are being transferred, the number of bytes is not quite as obvious. Each sample in the waveform or element in the array may require several bytes to transmit. For example, ASCII waveform data from a TEKTRONIX 7D20 Programmable Digitizer requires from 3 to 5 characters (bytes) per sample to transmit plus a comma delimiter. A typical 7D20 waveform is shown below.

CURVE 0.4,0.12,0.04,-0.08,-0.12,-0.64,-0.88,-1.12 ...

For a 1024-point waveform, between 4096 and 6144 bytes are transmitted for the waveform data, plus six bytes for the header (CURVE) and the space following the header. The exact number of bytes depends on the values of the individual samples. Some values require more characters (bytes) to send than others.

Binary waveforms only require one or two bytes per point plus the header, byte count, and checksum bytes. As a result, binary waveforms are usually faster to transmit since fewer bytes are involved.

Regardless of the format of the waveform, the data transfer time is simply the number of bytes divided by the data burst rate. Remember however, that the burst rate is limited to the rate of the slowest device, so the maximum burst rate of the 4041 may not be realized. In practice, the actual transfer rate is usually less than the maximum rate of either device.

Unaddressing Sequence—at the end of most data transfers, the 4041 sends the universal UNTalk and UNListen commands to make sure that the bus is left clear for the next transfer. This process takes about 1.7 milliseconds, assuming that no devices on the bus slow down the transfer.

Remember that WBYTE and RBYTE do not automatically send UNTalk or UNListen after a transfer. These bytes may be explicitly included in a WBYTE statement. See **WBYTE and RBYTE Timing** for more information on the time required to transfer these bytes.

An example—estimating data transfer rate for a PRINT statement. The following example illustrates how the six components of a data transfer fit together to form a complete data transfer. For the sake of this example, consider the following statements:

```
100    Open #10:"GPIB0(pri=10,tra=fas):"
110    Integer waveform(2048)
120    Dim a$ to 2052
         :
         :
200    Print #10 buffer a$ using "+8%":waveform
```

When line 200 begins executing, the first thing the 4041 does is begin the statement overhead period. The statement overhead period for this statement was estimated by measuring the time to execute the statement:

```
Print buffer a$ using "+8%":;
```

This estimate will be slightly low since the LUN included in the PRINT statement takes a little more time to interpret. However, the estimate is close enough for most purposes. In this case, the measured overhead time is about 6.8 milliseconds.

Once the addressing is complete, the 4041 loads the buffer string with data ready for output. This buffer overhead time can be measured by measuring the time to execute the following statement:

```
Putmem buffer a$ using "+8%":waveform
```

From Table 7-7, the buffer overhead for this statement should be about 35 microseconds per

point, or about 71.7 milliseconds. The measured time for the PUTMEM statement is 72 milliseconds.

For the sake of this example, assume that the device receiving this data is much faster than the 4041. Based on this assumption, the burst rate of the 4041 can be used as an estimate.

To determine the data burst time, simply divide the total number of device-dependent data bytes by the data burst rate. The binary block format specified in the USING clause (+8%) generates a total of 2052 bytes (1 block header character + 2 byte-count bytes + 2048 data bytes + 1 checksum byte). In FASt mode, the 4041 can send data at about 19.5 K bytes/sec, so the data burst time is:

Burst time (minimum) = number of bytes/transfer rate
= 2052/19,500
= 105.2 milliseconds

At the end of the transfer, the 4041 initiates the unaddressing sequence. This sequence takes about 1.7 milliseconds.

A summary of the complete estimate is shown below.

Statement overhead	≈	6.8 ms
Addressing sequence	≈	1.7 ms
Buffer overhead (2052x35 μs)	≈	71.7 ms
Data burst (2052/19,500)	≈	105.2 ms
Unaddressing sequence	≈	1.7 ms
Estimated	≈	187.1 ms

Estimating serial poll time. The serial poll operation is performed in most systems every time a device on the bus asserts SRQ. Though the process is relatively fast compared to large data transfers, the time may be significant enough in larger systems to require that it be included in performance estimates.

Figure 7-7 shows a flow chart of the serial poll operation as implemented by the 4041. Approximate times for each step are included on the flow chart as an aid in developing an estimate of the time required to perform a serial poll. Remember that these values assume that the instrument does not significantly delay the process.

Remember from the discussion of the POLL statement in Section 5 that there are three forms of POLL. One form polls a single device whose address is specified in the POLL statement. This form returns the status byte from the instrument regardless of whether the instrument is asserting SRQ or not. In this form, the loop shown in Fig. 7-7 is only executed once. The status byte is returned whether bit 7 is set or not.

The second and third forms of POLL are a little different. In the second form, a list of device addresses are specified in the POLL statement. The 4041 polls the first device in the address list and checks the status byte to see if bit 7 is set. If it is, the UNTalk and UNListen commands are sent and the statement terminates. The status byte and the address of the device are returned in the status and address variables.

If the first device in the list is not asserting SRQ, bit 7 of its status byte is not set, so the 4041 polls the next device in the list. This process is repeated until the 4041 either finds a device that is asserting SRQ (bit 7 is set) or the last device is polled. If no device in the list is asserting SRQ, the 4041 reports error number 815 or error number 816.

Notice that the number of iterations through the serial poll loop, and thus, the time required to complete the poll is dependent on which device is asserting SRQ.

The third form of the poll statement is similar to the second in that several devices may be polled. The only difference is that no address list is included in the POLL statement. Instead, the 4041 begins polling all addresses, starting with primary address 0. The 4041 waits for the amount of time specified by the GPIB SPE parameter for a response. If no device responds, the 4041 continues to the next address.

This process is repeated until a device is found that is asserting SRQ, or until all primary addresses have been polled. If no device is found, the 4041 tries again using all valid combinations of primary and secondary addresses, starting with primary address 0 and secondary address 0. If a device asserting SRQ still isn't found, the 4041 reports an 815 or 816 error.

If you are using the first form of the POLL statement, or if you know which device is asserting

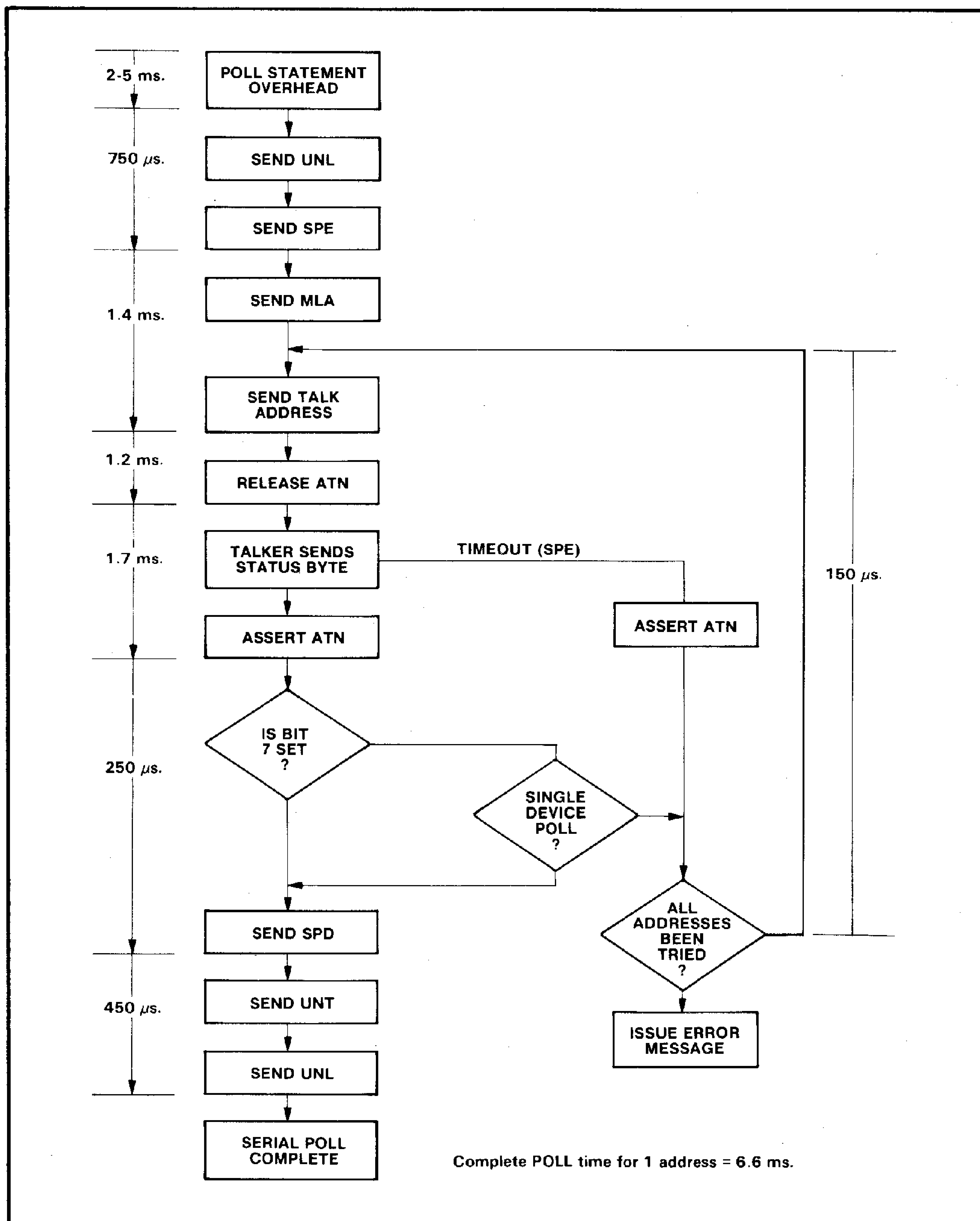


Fig. 7-7. *A flow chart of the POLL process. Approximate timing values are shown as an aid to estimating execution time.*

SRQ, you can make an estimate of the serial poll time using the timing values shown in Fig. 7-7. Keep in mind that most of the bus traffic involved in a serial poll process is interface messages, so all devices on the bus are involved in the handshake. As a result, if any of the devices are slower than the 4041, that device will limit the serial poll time.

For a POLL without an address list, the SPE parameter sets the amount of time the 4041 will wait for a device to respond before going on to the next address. Thus, the SPE value directly affects the time required to complete the poll.

WBYTE and RBYTE timing. The low-level WBYTE and RBYTE statements provide more direct control of the GPIB. These statements allow you to send interface messages as well as device-dependent bytes. The execution of WBYTE and RBYTE statements isn't divided as neatly into five components like PRINT and INPUT. The statement overhead still occurs, but each byte is processed and transmitted individually, so it's difficult to separate the overhead and data transfer time.

Generally, the WBYTE and RBYTE statement execution time must be considered as a unit. Table 7-9 shows some typical WBYTE and RBYTE statement execution times when transferring data to and from a very fast device.

TABLE 7-9
WBYTE STATEMENT OVERHEAD AND TRANSFER TIME

Statement	Execution Time
Wbyte dcl	7.8 ms
Wbyte dcl,spd	13.8 ms
Wbyte #1:atn(unt)	8.4 ms
Wbyte #1:atn(unt,mta)	12.8 ms
Wbyte #1:atn(unt,mta,34)	13.6 ms
Wbyte #1:get(1,2,3)	10.8 ms
Wbyte #1:"SET?"	8.2 ms
Wbyte #1:"SET?",eoi	8.8 ms
Wbyte #1:number	8.0 ms
Wbyte #1:number,number	14.2 ms

These values are valid only when the 4041 is talking to a device that is much faster than itself. Also, remember that all devices on the bus must handshake messages with attention asserted (interface messages), so the slowest device on the bus limits the transfer rate.

Table 7-10 shows the execution time for some typical RBYTE statements executed in Fast mode with a very fast talker. As in WBYTE, the various components of the I/O process are not easily distinguished. However, the values listed in the table provide a guideline for estimating the total execution time for RBYTE statements.

TABLE 7-10
RBYTE STATEMENT OVERHEAD AND TRANSFER TIME

Statement	Execution Time
Rbyte string$ (1 character)	7.2 ms
Rbyte string$ (10 characters)	9.0 ms
Rbyte string$ (100 characters)	26.4 ms
Rbyte array (1 element)	7.8 ms
Rbyte array (10 elements)	61.0 ms
Rbyte array (100 elements)	600.0 ms

Estimating data processing time. Once the data is acquired and transferred to the controller, some type of processing is often required to extract useful results. This processing may be done entirely in the controller, it may be shared between the controller and intelligent instruments, or the instruments may process the data and send final results to the controller. For example, the DM 5010 Programmable Digital Multimeter can perform some simple operations, such as limit comparison, averaging, and offset and scale conversion. The 7854 Programmable Oscilloscope is an example of an instrument that can execute its own measurement programs and return computed results to the controller.

The processing time is difficult to estimate. Often, the only practical way to estimate processing time is by direct measurement. Benchmark comparisons may be helpful for comparing various controllers, but they usually only compare speed in one type of operation. This may not be a valid comparison for your application. Also, if the processing burden is shared between intelligent instruments and the controller, some processing may be overlapped with other tasks so that you can do more in the same amount of time.

Many controllers have some type of real-time clock that can be used to measure the execution time for a particular task. Remember that in some cases, such as sorting routines, the processing time depends on the input data. In these cases, it's important to measure the processing time under conditions as close to the real operating environment as possible.

Some general hints for using the real-time clock in the 4041 to make timing measurements are provided later in this section.

Estimating human interaction time. Probably the most difficult parameter to estimate in any system is the time required for operator interaction. As with data processing, direct measurement is often the only feasible way to estimate human interaction time. Suggestions for minimizing human interaction are included in Section 8.

Using the 4041 Real-Time Clock for Timing Measurements

The 4041 includes a real-time clock that allows you to keep track of the time and date, as well as make timing measurements. The Ask("TIME") function returns the number of seconds since power-up. The value returned has a 10-millisecond resolution. As a result, for most tasks, you can directly measure execution time by putting an Ask("TIME") function before and after the code as shown in Fig. 7-8.

```
100    Time1=ask("TIME")
       :
       Rem ** Insert process to
       Rem ** be measured here.
       :
300    Time2=ask("TIME")
310    Exectime=time2-time1
```

Fig. 7-8. *The time to execute a program or segment of a program can be measured using the 4041 real-time clock.*

In some very short operations, the 10 millisecond resolution may not be sufficient. In these cases, you can perform the operation repetitively in a loop, measure the total time, and divide it by the number of iterations. Some overhead is added to the execution time by the loop statements (GOTO or FOR/NEXT), but you can measure this loop overhead and subtract it from the execution time. To measure the loop overhead, simply execute an empty loop the same number of times as the main process loop and measure the time required. Then, subtract this overhead from the execution time for the test loop (Fig. 7-9).

```
100    Time1=ask("TIME")
110    For i=1 to loops
120     Next i
130    Time2=ask("TIME")
140    For i=1 to loops
       :
290     Next i
300    Time3=ask("TIME")
310    Exectime=(time3-time2-(time2-time1))/loops
```

Fig. 7-9. *The effective resolution of the real-time clock can be improved by repeating the execution of the measured program in a FOR loop. Since the FOR/NEXT statements add some overhead, this overhead is separately measured and subtracted from the measured time. The results are divided by the number of loops.*

Section 8—Improving 4041 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 much each component contributes to the overall performance picture. Section 7 describes each of the components and provides some ideas for estimating the time required for each component.

Developing an estimate of system performance also has another benefit—it helps you identify the bottlenecks in system performance. In many cases, one or two of the factors dominate the overall performance. The obvious place to start making speed improvements is in these bottlenecks. If your system spends a large part of its time transferring data, start by making improvements in the data transfers. If, on the other hand, the system spends most of its time processing data and interacting with the operator, data transfer improvements probably won't buy much in overall performance gain. It doesn't do much good to trim 50% off a task that only consumes 5% of the system's time.

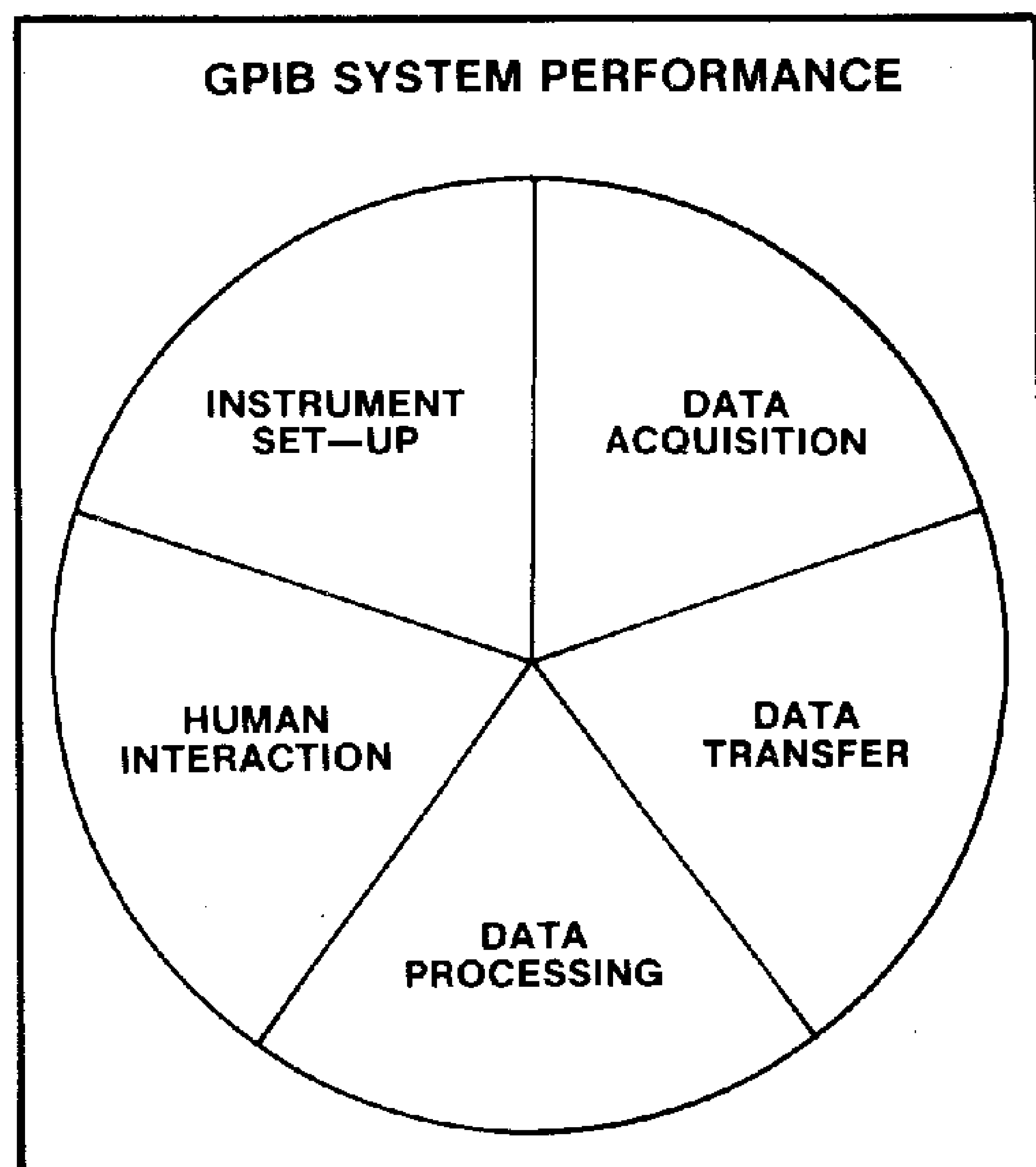


Fig. 8-1. *The GPIB performance picture is composed of five major parts. The relative amount of time spent on each component varies with different systems.*

Some suggestions for speed improvement are provided here for each of the five components of system performance (Fig. 8-1). This certainly doesn't represent an exhaustive list of improvements, nor is it a "cure-all" for all situations. But, it does provide a starting point for "tuning" the system for best performance.

Get to Know the System

Getting the most from any system requires that you become very familiar with the controller and instruments. There's no substitute for careful study of the documentation. In particular, you'll need to know the command vocabulary for the instruments as well as the data formats required. If several formats (e.g., ASCII and binary) are acceptable, decide which format is the fastest and most efficient for your application.

You'll also need to know how the instrument buffers and executes commands. Can the instrument receive multiple commands in a single message? What happens if you send commands over the bus while the instrument is busy executing other commands? Does the instrument respond to requests for data while an acquisition is in progress?

These are just a few of the things you need to learn about your instruments and controller. The better you know the features and capabilities of your instruments, the better you'll be prepared to write efficient programs. The Operators and Programmers manuals are your best source of information. In addition, a brief interfacing guide called an IIG (Instrument Interfacing Guide) is provided with Tektronix programmable instruments. A little time spent reading this documentation can save hours of programming problems and frustrations.

Reducing Set-up Time

The key to improving set-up time is to either reduce the number of setting changes or reduce the time required for the instrument to execute the setting changes. Try these suggestions:

- **Reduce the number of setting changes.** If the application requires a large number of setting changes, you can reduce the number of changes by grouping the tests that use common settings together.

• **Set ranges explicitly.** Some instruments, particularly DMMs, provide an auto-ranging feature that allows the instrument to automatically adjust its range to the input signal. However, the auto-ranging process is usually slower than explicitly changing the settings. As a result, when speed is a critical factor, it's usually better to use explicitly set ranges instead of the auto-ranging function.

• **Set up slow instruments first.** If some instruments in the system require settling time, set these instruments up first so that they can settle while other devices are being set up.

• **Use internal setting storage.** If your instrument provides a facility for storing instrument settings locally, this is often the fastest way to make setting changes. The savings are two-fold. First, the command to recall stored settings is usually shorter (and thus faster to transmit and execute) than equivalent setting commands. Second, the instrument doesn't have to interpret the individual commands because the settings are usually stored in a decoded form that can be directly executed.

For example, the TEKTRONIX FG 5010 Programmable Function Generator can store up to ten complete instrument set-ups (except for the STEP SIZE, DT, PLI, RQS, and USER settings). Any one of these complete instrument set ups can be retrieved using the RECall command.

• **Use low-level setting commands.** Some Tektronix instruments also provide an alternate "low-level" command set that is more complex, but faster to execute. This feature allows you to transmit and read settings in a pre-processed form. Since the settings are pre-processed, the instrument decodes and executes them faster. However, on some instruments this feature can only be used to transfer complete instrument set-up information, not individual control settings.

For example, the low-level settings command (LLSET) for the FG 5010 Programmable Function Generator allows you to send and receive all the instrument settings in a binary block.

Reducing Data Acquisition Time

Reducing data acquisition time requires careful attention to detail in setting up the acquisition. Though it's often impossible to change the input signal, you can take a few steps to make data acquisition faster.

• **Use sequential digitizing.** The choice of sequential or equivalent-time digitizing involves many factors. Often, the choice is dictated by available equipment. However, when a choice is available, sequential digitizing is usually faster than equivalent-time digitizing. (The differences between sequential and equivalent-time digitizing are described in Section 7.)

A few digitizers, like the TEKTRONIX 7D20 Programmable Digitizer offer both modes. At lower TIME/DIV settings, the 7D20 uses sequential sampling. It automatically switches to equivalent-time digitizing at higher TIME/DIV settings.

• **Reduce the trigger delay.** In some applications, input signal parameters can be controlled. In these cases, you can reduce the trigger delay by increasing the repetition rate of the signal. This is particularly effective when using an equivalent-time digitizer, since many triggers are required to acquire the waveform.

If the repetition rate cannot be controlled, careful selection of a trigger mode may improve performance. Consider, for example, acquiring readings with a DMM. Most DMMs can acquire samples in free-run mode or triggered mode. In free-run mode, the DMM takes samples continuously—no trigger is required. In triggered mode, the DMM requires a trigger to capture a sample.

The choice of which mode to use depends on your application. If the reading must be synchronized with an event or other system operation, triggered mode is probably required. However, if the reading doesn't have to be synchronized, the free-run mode assures that a reading is always available. If triggered mode is used, be sure the proper trigger is supplied as soon as possible.

Also remember that pre-trigger samples in a digitizer or logic analyzer must be acquired before the trigger can be accepted. Use pre-trigger mode only when necessary and be sure you know how the instrument acquires pre-trigger data.

• **Reduce the digitizing time.** When you are using a sequential digitizer, the digitizing time is directly related to the sample interval, as discussed in

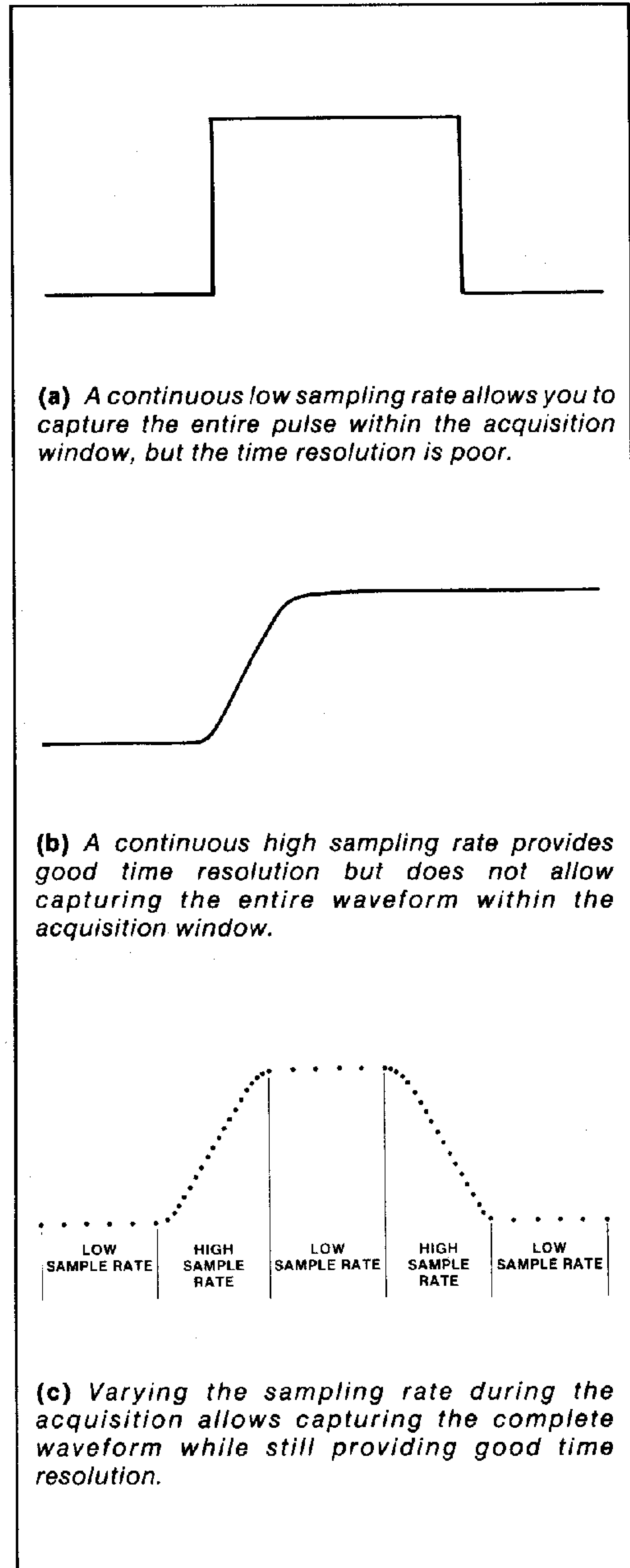
Section 7. Digitizing time is simply the number of sample intervals (the number of samples minus one) times the sample interval. Or, if the digitizer's time base is calibrated in time per division, the digitizing time is the time per division times the number of horizontal divisions.

There are only two variables in the equation—sampling interval and the number of samples. As a result, the only options for reducing data acquisition time are reducing the number of samples captured or decreasing the sampling interval. If the digitizer does not allow you to set the record length (number of samples), adjusting the sampling interval may be your only option.

A few digitizers, such as the TEKTRONIX 7612D Programmable Digitizer and the SONY/TEK 390AD Programmable Digitizer allow you to change the sampling interval "on the fly" during an acquisition. For example, with the 7612D you can use a short sampling interval (high sampling rate) during the fast rise of a pulse. Then, you can use a slower rate during the flat top of the pulse, and switch back to the fast rate for the falling edge (Fig. 8-2). This eliminates the time required for making two or three acquisitions to capture the entire pulse and still achieve high time resolution.

With equivalent-time digitizers, the digitizing time depends on both the sweep speed and the repetition rate of the input signal. Higher sweep speeds mean fewer samples per repetition and thus, more repetitions are required to complete an acquisition. Lower repetition rates mean you spend more time waiting for the trigger to initiate each sweep. The best sweep speed is a compromise between the better time resolution offered by higher sweep rates and the faster acquisition afforded by lower sweep rates.

Faster digitizing can be achieved by increasing the repetition rate of the input signal (if possible), or by decreasing the sweep speed. If the acquisition prohibits either of these techniques, one solution is to use a sampling plug-in like the 7S12 TDR/Sampling Unit. The 7S12 samples many repetitions of the input signal at very high speed and reproduces the signal at a much lower frequency for input to a digitizer, such as the 7854 Programmable Oscilloscope. Using this technique, the 7854's real-time sweep speed can be set much lower than would be required to capture the signal with a



(a) *A continuous low sampling rate allows you to capture the entire pulse within the acquisition window, but the time resolution is poor.*

(b) *A continuous high sampling rate provides good time resolution but does not allow capturing the entire waveform within the acquisition window.*

(c) *Varying the sampling rate during the acquisition allows capturing the complete waveform while still providing good time resolution.*

Fig. 8-2. *A digitizer that allows changing the sampling interval during acquisition allows you to capture a waveform in a single acquisition that might otherwise require three or more acquisitions.*

conventional time base and amplifier. The result is faster acquisition.

• **Use SRQ interrupts to boost performance.** A significant amount of time can be wasted in a program waiting for an acquisition to complete. One way to take advantage of this "wasted" time is to use the operation-complete or waveform-readable interrupt available on most Tektronix programmable instruments.

A handler routine can be set-up to detect the operation-complete interrupt. Using this technique, the 4041 initiates an acquisition and continues with other processing. When the acquisition completes, the instrument generates an SRQ interrupt and reports operation-complete status. The 4041 calls the SRQ handler routine which reads the new data and sets a flag variable to indicate to the main program that new data is available.

Section 5 describes the use of the operation-complete interrupt in more detail and provides program examples.

Reducing Data Transfer Time

GPIB data transfer improvements can be made in two basic areas. First, the system configuration can affect data transfer speed. And second, a number of improvements can be made to the programs that control the data transfer.

• **Choose the bus configuration carefully.** One obvious way to improve data transfer rate is to use instruments that are faster on the bus. But be careful! Remember that data transfer rate is limited by the slowest device involved in a transfer. It doesn't do any good to get a faster instrument or controller if other devices involved in the transfer limit the transfer rate.

If your controller supports more that one GPIB port, think carefully about how you divide the instruments on the two buses. Avoid separating instruments that regularly interact, such as a digitizer and a tape drive. If the instruments are on the same bus, you may be able to transfer data directly from one to the other without going through the controller (assuming that their data formats are compatible). If the instruments are separated on two buses, the data will have to pass through the controller.

If one of the GPIB ports on the controller is faster than the other (e.g., one supports Direct Memory Access), it's best to put the faster instruments and the ones that transfer the largest amounts of data on the faster port. On a 4041 with the Option 1 GPIB port, put the fastest instruments or instruments that generate large amounts of data on the second port (GPIB1).

It usually isn't necessary or helpful to segregate slower instruments to a different bus since the transfer rate is limited only by the devices involved in a transfer. Unaddressed devices do not affect the transfer rate.

Also, if one of the ports is used to drive a device-under-test, put the instruments used to perform the test on a different bus from the device-under-test. That way, if the device-under-test halts bus transfers due to an error or malfunction, the test instruments won't be affected by the error. Figure 8-3 illustrates a system configured in this manner.

• **Choose the right I/O statements.** The first step to writing efficient GPIB data transfer programs is to become familiar with the controller's GPIB I/O statements. In 4041 BASIC, most transfers are done with PRINT, INPUT, WBYTE, or RBYTE. In most cases, PRINT and INPUT statements are the best choice, since they execute faster than WBYTE and RBYTE. One exception is string input. For long strings, RBYTE is faster than INPUT. However, since RBYTE does not automatically generate an addressing sequence, the devices involved in the transfer must be addressed with WBYTE, which is slower than PRINT.

PRINT and INPUT can be used to transfer most types of data with the USING clause and the delimiters (EOM, EOH, EOU, etc.). Data formats that are not supported by USING can still be read into a string variable by setting the EOM character to null (0). Then, the GETMEM statement or VAL, VALC, and SEG$ functions can be used to parse the string. For example, you can transfer binary data into a string variable with EOM set to 0. Then, GETMEM or VAL can be used to parse the data out of the string variable and put it in numeric variables.

WBYTE and RBYTE are most useful when you need more direct control of the GPIB or when handling data formats that aren't supported by the

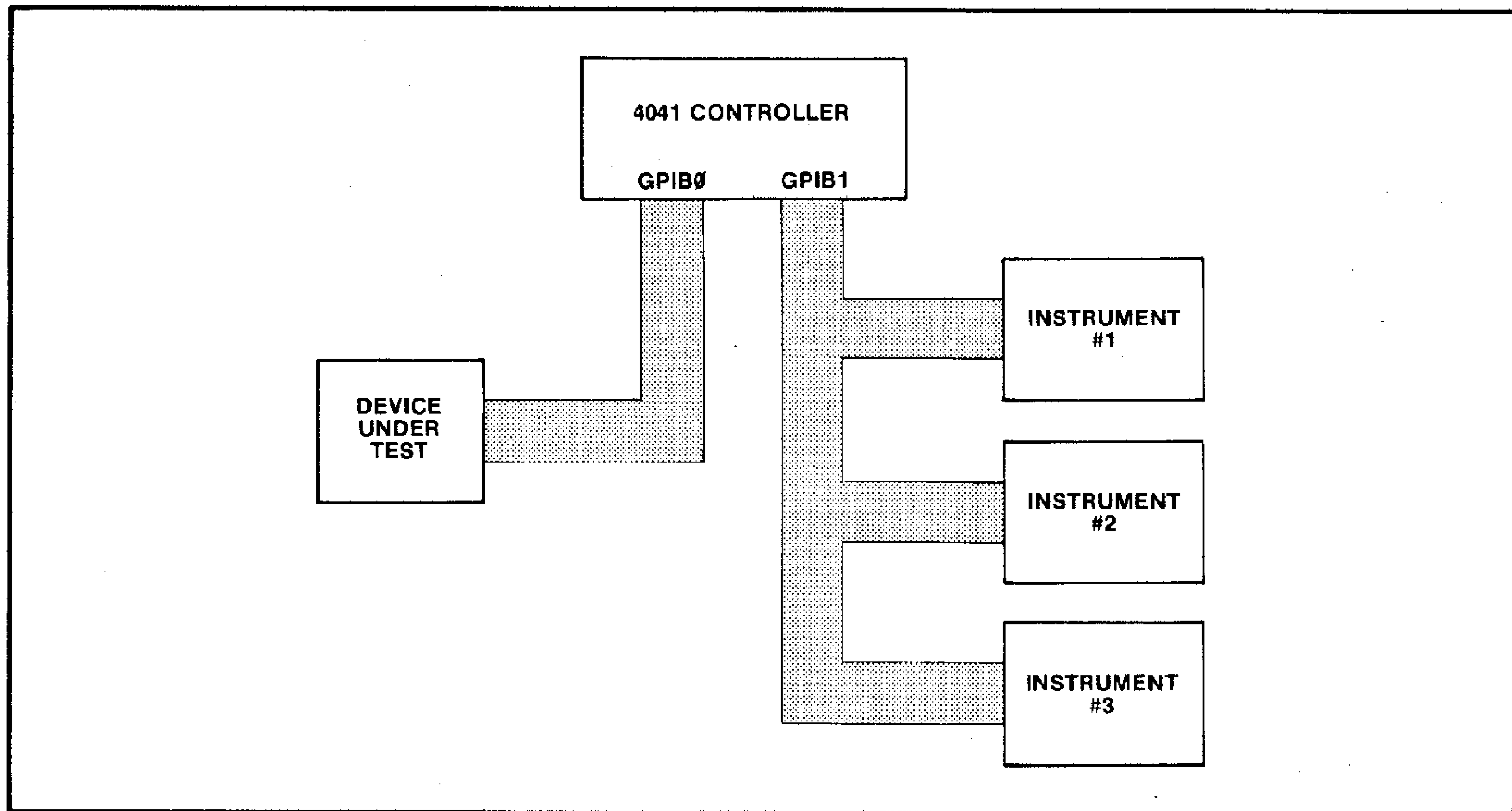


Fig. 8-3. *Putting the device under test on a different bus from the test instruments prevents D.U.T. failures from affecting the test system operation.*

USING clause. For example, with WBYTE, you can set-up a transfer directly between two instruments. WBYTE will send addresses as well as other interface messages. RBYTE is slower than INPUT for numeric data but RBYTE is faster than INPUT for strings.

- **Minimize bus traffic.** If you can't make bytes transfer across the bus faster, the other way to gain performance is to reduce the number of bytes transferred. There are a number of ways to reduce the bus traffic without degrading the integrity of the messages.

Whenever possible, use the shortest form to send data. When specifying the number of characters or digits in a USING clause, include only the required characters. Extras are padded with zeros or spaces. The "F" modifier for USING tells the 4041 to use only the required number of digits or spaces, eliminating any extra characters.

- **Abbreviate command names.** The command names for Tektronix programmable instruments are intended to represent the command function as clearly as possible. For example the FREQUENCY command sets the frequency of a function generator. However, most of these commands can also be abbreviated to a much shorter form—sometimes as short as one or two letters. Using the shortened form saves a little bus traffic, but there is also a cost. Longer, more readable command names make program maintenance much easier than cryptic single-letter codes. The best compromise between readable code and fast data transfers depends on your system. Unless a large part of the system time is spent transferring commands (which is usually not the case), shortening the command names won't affect overall performance significantly.

- **Minimize buffer overhead.** All GPIB I/O in the 4041 passes through a special area of memory called the I/O buffer. This buffer is described in more detail in Sections 4 and 7. The I/O buffer is used as a "staging area" for transfers. If the amount of data transferred is too large to fit in the I/O buffer, the transfer proceeds in blocks the size of the I/O buffer.

The time required to move data in and out of the I/O buffer is called buffer overhead. One way to reduce buffer overhead is to make the I/O buffer

larger so that the entire data transfer can fit in the buffer. The BUFFER clause in 4041 BASIC allows you to define a string variable as an I/O buffer. The size of the I/O buffer is then determined by the dimensioned length of the specified string variable.

The buffer overhead time can even be eliminated almost entirely if the data can be stored directly in a string variable. For example, assume that a waveform is being transferred from a 7D20 Programmable Digitizer. For the sake of this example, assume that the data will only be stored on tape for later processing. The following program segment illustrates how the buffer overhead time can be eliminated.

```
100  Dim buffer$ to 6144,dummy$ to 1
110  Input #10 prompt "CURVE?" buffer buffer$:dummy$
120   Rem ** Store the data in BUFFER$ on tape. **
       .
       :
```

In this program, an I/O buffer string (BUFFER$) is dimensioned large enough to hold an entire ASCII waveform. A dummy string (DUMMY$) is also dimensioned to 1 character. The INPUT statement gets the waveform, using BUFFER$ as an I/O buffer. As a result, BUFFER$ contains exactly what the instrument sent, including the CURVE header and the ASCII waveform data.

Since the destination string (DUMMY$) is only one character long, the 4041 puts the first character from BUFFER$ in DUMMY$, and then stops. The contents of BUFFER$ are unmodified and still available for use. The dummy string is required only to maintain correct syntax for the INPUT statement.

Since only one character was moved from the I/O buffer, the overhead is practically eliminated. This can cut the total transfer time significantly in many cases.

• **Use string variables when possible.** String variables can be used as buffer strings, as previously discussed. But, using them as the source and destination variables in a PRINT or INPUT statement can also reduce buffer overhead time. That's because string variables store data directly from the I/O buffer in most cases. Little or no data formatting is required, so the process of moving data between variables and the I/O buffer is faster.

For example, transferring a 512-point ASCII waveform plus the waveform preamble from a 7854 Programmable Oscilloscope takes about 1.02 seconds when the data is stored in a string variable. The same transfer takes about 2.04 seconds when data is stored in a numeric array. The difference is the additional overhead required to convert the data from the ASCII format in the I/O buffer to internal storage format for numeric data.

• **Choose the data transfer mode carefully.** 4041 BASIC provides two data transfer modes (Normal and Fast) for the standard GPIB port (GPIB0) and three modes (Normal, Fast, and DMA) for the Option 1 GPIB port (GPIB1). The choice of which mode to use depends on your application. A basic understanding of each mode will help clarify which is best for your application.

NORMAL TRANSFER MODE: In normal mode, the 4041 services GPIB data transfers via interrupts from its GPIB interface. Each time a byte is transferred over the GPIB, the interface generates an interrupt that signals the 4041 to prepare another byte for transfer.

FAST TRANSFER MODE: In Fast mode, the 4041 services GPIB data transfers by continuously checking the GPIB interface to see if it is ready to transfer another byte. This eliminates the delay required in normal mode to stop other processing and respond to an interrupt. Proceed mode may be used with Fast transfer mode, but program execution does not proceed until the I/O is finished. Thus, Proceed mode, though valid, is not effective in Fast transfer mode.

DMA TRANSFER MODE: The Option 1 GPIB interface (GPIB1) has a third mode, called Direct Memory Access (DMA) mode. In this mode, the 4041 tells the GPIB interface where to store data for input or where to get data for output. The interface transfers data directly between the bus and memory without involving the main processor at all. This mode is considerably faster, but it can only be used on the Option 1 interface.

• **Use Proceed mode.** In Normal and DMA transfer modes, the 4041 also provides a special mode, called Proceed mode, that allows you to overlap data transfers with other tasks. In proceed mode, an I/O operation can be in process while the 4041 proceeds with other operations. For example, consider the following program segment:

```
100     Set Proceed 1
110     Dim waveform(1024)
120     Print #1:"CURVE ";waveform
130     .
        :
```

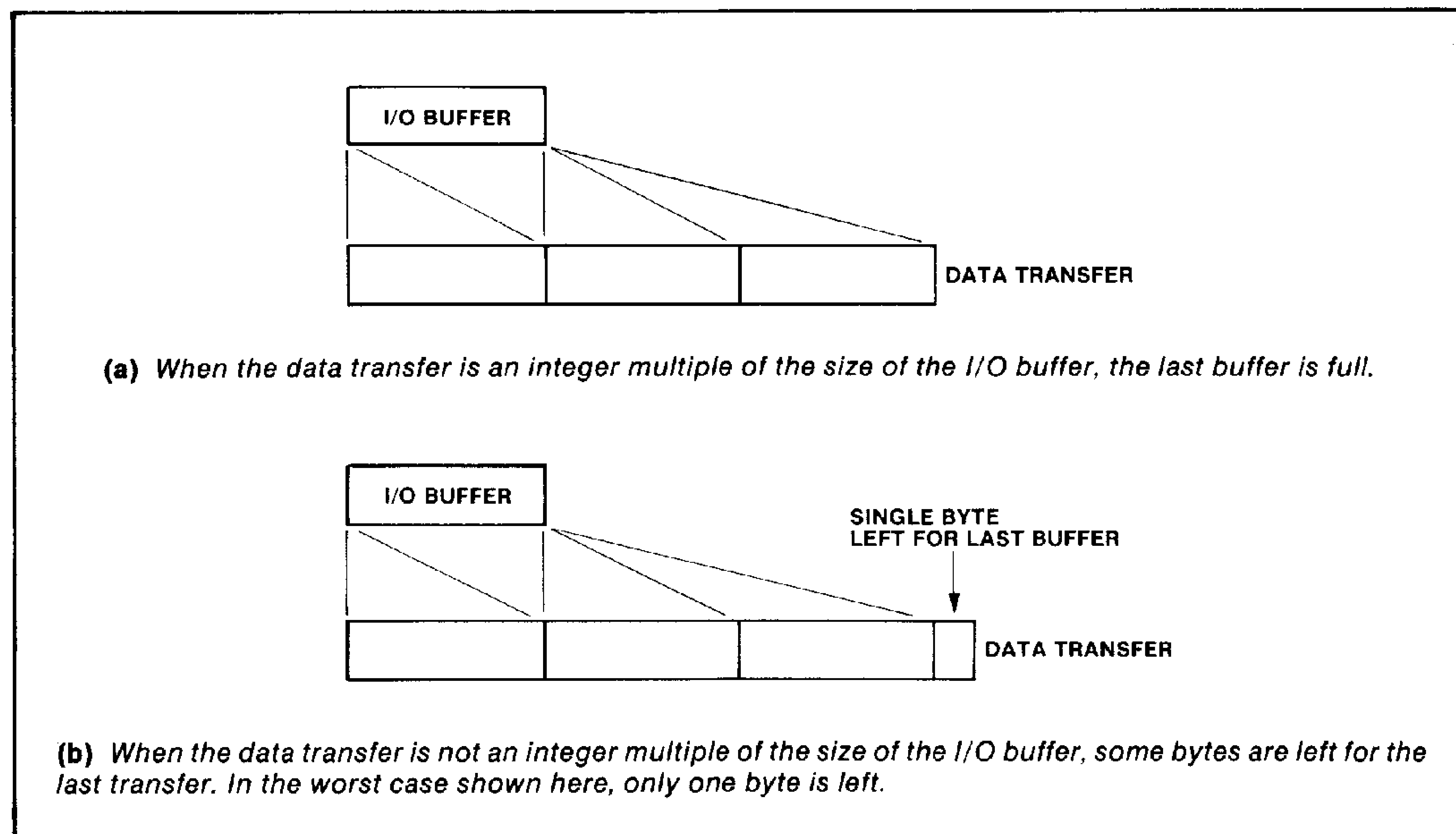
The 4041 is set to proceed mode in line 100. When the PRINT statement in line 120 is executed, the 4041 waits just long enough to load the I/O buffer and initiate the output process. Then, execution continues with line 130. Notice that the 4041 does not proceed to the next statement until the last data is loaded in the I/O buffer. As a result, if the I/O buffer is not big enough to hold the entire data transfer, execution does not continue until the previous blocks are transmitted and the last block is loaded into the I/O buffer.

When the size of the data transfer is not an integer multiple of the I/O buffer size, the last buffer load could be small. In the worst case, where one byte of data is left for the last buffer, execution does not proceed until the last byte is loaded in the I/O buffer, so proceed mode doesn't buy much time.

Figure 8-4 illustrates the relationship between the size of the I/O buffer and the number of bytes to transfer. Part **a** of the figure shows the case where the number of bytes to be transferred is an integer multiple of the buffer size. The last buffer is full and execution proceeds as soon as the last buffer is loaded. Part **b** of the figure shows the worst case where only one byte is left for the last buffer. Execution doesn't continue until this last byte is loaded in the I/O buffer.

Also, remember that if the processing that follows an INPUT or RBYTE statement uses variables being filled by the INPUT or RBYTE statement, execution can't continue until the transfer is complete. Execution is automatically delayed at the point where the data is referenced.

The key to using Proceed Mode effectively is in choosing when to use it and in choosing the I/O buffer size. Don't use proceed mode when the program execution requires immediate access to the data involved in the transfer. In these cases it's better to use the FAST or DMA transfer mode to just speed up the transfer.



(a) *When the data transfer is an integer multiple of the size of the I/O buffer, the last buffer is full.*

(b) *When the data transfer is not an integer multiple of the size of the I/O buffer, some bytes are left for the last transfer. In the worst case shown here, only one byte is left.*

Fig. 8-4. *Execution does not proceed to the next statement in Proceed mode until the last data is loaded in the I/O buffer. If only a few bytes are left for the last block, as in part* **b**, *execution doesn't proceed until the transfer is within the last few bytes of completion.*

Use an I/O buffer that is as large as the entire block to be transferred. If memory restrictions don't allow a buffer this large, use the largest possible buffer that divides the transfer into equal size blocks. This will make the last block as large as possible, allowing the execution to proceed at the earliest possible point in the program.

4041 BASIC also provides an IODONE interrupt that can be used to detect the completion of a Proceed-mode I/O operation. The IODONE interrupt condition and its use is described in Section 5.

• **Use binary data.** If your instruments can send data in binary format as well as ASCII, it may be advantageous to use binary instead of ASCII. Binary data is a little more complicated to handle than ASCII, but binary transfers are usually much faster since they involve fewer bytes than an equivalent ASCII transfer.

Each numeric value in ASCII requires one byte per digit in the numeric value. Thus, if a 6-digit numeric value is transmitted, six bytes must be transferred. In binary format, the numeric value is encoded directly into one or two bytes, instead of sending the ASCII code for the digits. Figure 8-5 illustrates the difference.

The difference between the two transfer modes can be pretty dramatic. Table 8-1 shows the time required to transfer a 1024-point waveform from a 7D20 to a 4041 in ASCII format and binary format. Both transfers were done in DMA mode into a large buffer string.

TABLE 8-1
7D20/4041
TYPICAL WAVEFORM TRANSFER TIME

Data Format	Transfer Time
Binary	0.20 sec.
ASCII	1.38 sec.

Two binary formats are defined by Tektronix Standard Codes and Formats. Tektronix programmable instruments that transfer binary data use these formats. The 4041 provides operators for the USING clause that handle these formats automatically. Section 4 describes the USING clause and the binary block operators.

Reducing Data Processing Time

If your system does a lot of data processing, you may need to find some ways to get the data processing done faster. There are three basic areas where improvements can be made. First, the algorithms can be changed to reduce the required processing. Second, the processing can be made faster by using ROM routines which execute faster than BASIC programs. And finally, the processing burden can be shared among the controller and intelligent instruments.

• **Use a fast algorithm.** Finding the fastest way to perform an operation can be difficult. Entire books have been published on algorithms for some common operations. A notable example is sort algorithms. The simplest "bubble" sort algorithm is two to twenty times slower (depending on the amount of sorting required) than a more complex "shell" sort. The point is, when speed is critical, carefully evaluate the algorithms used for time-intensive operations.

• **Use implied array operations instead of FOR loops.** Many processing operations involve numeric arrays of some sort. When performing any type of mathematical operation on an array, the processing is usually done on an element-by-element basis (see Section 6 for more detail). 4041 BASIC allows you to perform most array operations without explicitly stepping through all the elements of the array one at a time.

For example, if you are adding a constant to all elements of an array, you don't have to set-up a FOR loop and add the constant to each element one at a time. Instead, you can just add the constant to the array without specifying an array index. The 4041 will automatically add the constant to all array elements in a single BASIC statement (Fig. 8-6). This is much faster than the FOR loop technique. The same holds true for most numeric functions, such as SIN and ABS.

Implied array operations have one disadvantage—they consume more memory than explicit FOR loops. 4041 BASIC creates temporary arrays to perform the implicit operation and then deletes the arrays when the operation is complete. Though some extra memory is required, implied array operations are much faster than explicit FOR loops.

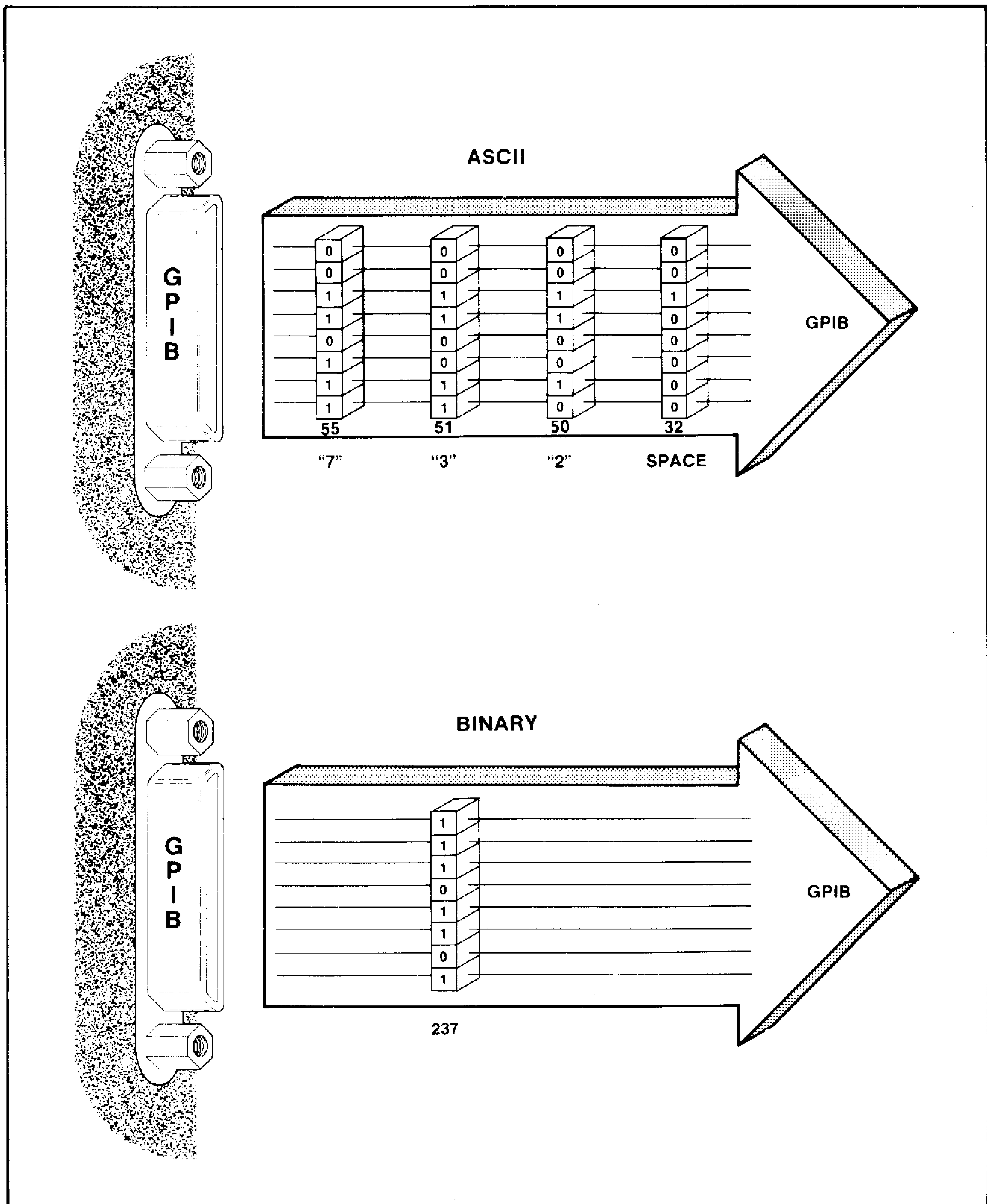


Fig. 8-5. *Transferring a value in ASCII requires one byte per digit, while transferring the same value in binary requires only one or two bytes.*

```
100    Constant=4
110    Dim array(100)
120    For i=1 to 100
130    Array(i)=array(i)+constant
140    Next i
```

a. *Performing the operation with a FOR loop.*

```
100    Constant=4
110    Dim array(100)
120    Array=array+constant
```

b. *Performing the operation with an implied array operation.*

Fig. 8-6. *Performing an array operation is faster using an implied loop in 4041 BASIC than using an explicit FOR loop. The implied loop also makes a simpler, more compact program.*

- **Select the data type carefully.** 4041 BASIC allows you to store numeric data in one of three formats: integer format, short floating-point format (the default) and long floating-point format. These data formats can be mixed in a mathematical operation. For example, a program might add an integer value to a long floating-point value. When different types of data are used in an operation, it is called a mixed mode operation.

In any mixed mode operation, the values are converted to a common data type before performing the operation. If the type of the destination variable is explicitly specified, the result is converted to that type for storage. These data conversions take time and can cause round-off errors. As a result, you should avoid mixed-mode operations as much as possible—particularly when dealing with array data. Also note that all transcendental functions (SIN, COS, etc.) are computed in long floating-point format, regardless of the source data type, so using long-floating point data for these operations eliminates the conversion time.

- **Use ROM Pack routines.** A variety of ROM packs are available for the 4041 that provide common signal processing, graphics, and utility functions. For example, the signal processing ROM pack (4041R03) provides routines to find the minimum or maximum of an array, integrate an array, find the mean of an array, etc. The graphics and plotting ROM packs (4041R01 and 4041R02, respectively) provide low-level graphics primitives as well as high-level plotting calls for generating graphs and charts.

All of these routines are written in assembly language for the 4041's main processor, so they run much faster than equivalent BASIC programs. Also, the ROM routines consume less memory, so there is more room left for programs and data.

- **Share the processing burden.** Another way to reduce the time required for data processing is to share the processing burden between the controller and intelligent instruments. For example, the DM 5010 Programmable Digital Multimeter provides some simple calculation capability to compare measurements to preset limits, to average readings, or to scale readings to other units using pre-defined scaling constants.

The 7854 Programmable Oscilloscope can store and execute its own measurement programs independently of the controller. It can acquire data, perform complex processing operations, including decision making and branching capability, and send partially or fully processed results to the controller. This can significantly improve throughput by allowing a single controller to handle several instruments.

In addition, since the 7854 can send a processed result—often a single value, instead of a complete raw waveform—the data transfer time is reduced. 7854 programs can be down-loaded from the 4041 or entered from the 7854 Waveform Calculator keyboard.

Reducing Human Interaction Time

Human interaction time can be one of the hardest components of system performance to improve. The best bet is always to avoid human interaction if possible. But when it's unavoidable, there are a few things you can do to reduce the time required.

- **Use programmable interfaces and switchers.** One of the major parts of operator interaction is routing signal connections and moving parts on a production line. If the controller can be given control of these functions, human interaction time can be reduced. A variety of programmable switchers, such as the Tektronix SI 5010 Programmable Scanner

and 1360P/1360S Programmable Signal Multiplexer, are available for signal routing. In addition, the MI 5010 Multi-function Interface from Tektronix allows you to automate control and switching functions from the bus without developing custom interfaces. The MI 5010 accepts a variety of cards including relay scanners, digital I/O cards, digital-to-analog and analog-to-digital cards, etc. These cards can be installed in the MI 5010 in any combination to produce a "custom" interface to the device-under-test.

• **Simplify the user interface.** When human interaction is unavoidable, the key to making it fast is making it simple. Use graphics, menus, or other simple prompting aids to help the user interact as quickly and efficiently as possible. For simple input, a small user-definable keypad, such as the one on the 4041 front panel, is often better than a full-size terminal keyboard.

In addition, be sure to include complete error handling facilities in the program so that unexpected operator inputs don't "crash" the system. The time saved by not re-starting the system when operators make errors more than compensates for the few extra steps required to check user input and handle error conditions.

• **Overlap human interaction and processing with Proceed Mode.** Using Proceed Mode, the 4041 can continue with other processing while waiting for operator input. This allows the program to take full advantage of the time, instead of wasting time waiting for the operator. Obviously, the processing done during this time cannot depend on the operator input data.

Appendix A—Subsets Describe 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— Tek Standard Codes and Formats Waveforms

The Tektronix Standard Codes and Formats specifies a common format for waveform transfers from Tektronix instruments. The common format makes handling waveform data easier and more consistent. This section describes the Tektronix standard format and provides examples from several Tektronix digitizers and digitizing oscilloscopes.

There are two distinct parts to a Tek Codes and Formats waveform: the preamble and the curve. When they are sent together, the preamble always precedes the curve.

PREAMBLE: Contains data that describes the waveform such as waveform size, scaling information, format specifications, and auxiliary information, such as identification strings, etc.

CURVE: Contains the curve or waveform data itself. When combined with the preamble information, the data can be interpreted with appropriate units, scaling, and attributes.

An instrument may transmit the preamble and curve separately or together. Many instruments provide separate commands to elicit the preamble and the data or both. Some older instruments omit the preamble entirely.

The Waveform Preamble

The preamble consists of a header string (WFMPRE) followed by a series of arguments separated by commas. Each argument contains a mnemonic that identifies the parameter and the value of the parameter, separated by a colon. Examples are shown later in this section. Table B-1 shows the items included in the preamble.

TABLE B-1
WAVEFORM PREAMBLE CONTENTS

Preamble Argument	Mnemonic	Default Value
Waveform identification	WFID	Null String
Curve data encoding	ENCDG	ASCII
Number of points	NR.PT	Unspecified length
Point format	PT.FMT	Y
X multiplier	XMULT	1
X increment	XINCR	1
Point offset	PT.OFF	0
X origin offset	XZERO	0
X value offset	XOFF	0
X unit	XUNIT	Null string
Y multiplier	YMULT	1
Y origin offset	YZERO	0
Y value offset	YOFF	0
Y unit	YUNIT	Null string
Z multiplier	ZMULT	1
Z origin offset	ZZERO	0
Z value offset	ZOFF	0
Z unit	ZUNIT	Null string
Binary data field width*	BYT/NR	1
Binary number format	BN.FMT	LF
Binary data precision	BIT/NR	8
Curve error check	CRVCHK	CHKSM0
Byte error check	BYTCHK	NONE

* *The remainder of the parameters apply only to binary curve data; they are omitted when ASCII curve data is sent.*

Waveform identification (WFID). The waveform identification is a character string that is sent to identify the waveform source or other information. Individual instruments send different strings for the waveform identification.

Curve data encoding (ENCDG). Curve data encoding specifies whether the curve is transmitted in ASCII or in binary form. ASC or ASCII indicates an ASCII waveform; BIN or BINARY indicates a binary waveform.

Number of points (NR.PT). Number of points specifies the number of data points in the waveform data. Notice that this is often not the same as the number of characters transmitted, since a single data point may be represented by several characters.

Point format (PT.FMT). The point format defines the type of data represented by the curve data. The most common is Y format, where each data point represents a vertical value, such as input voltage to a digitizer, etc. Other formats are XY, where both the X (horizontal) and Y (vertical) values are transmitted for each point, YZ, where both the Y and Z (intensity or other third axis value) are transmitted, XYZ where X, Y, and Z values are transmitted for each point, and ENV where two Y values are transmitted for each point—an upper limit and a lower limit.

Axis specification. The axis specification arguments include XMULT, XINCR, PT.OFF, XZERO, XOFF, YMULT, YZERO, YOFF, ZMULT, ZZERO, and ZOFF parameters. These parameters define the scaling, origin, and offset of each data point. The values can be used in conjunction with the curve data to compute he actual X, Y, and Z value

at every point. The equations for computing each point are:

For Y format data:
X_n = XZERO + XINCR * (point number - PT.OFF)
Y_n = YZERO + YMULT * (data point - YOFF)

For XY format data:
X_n = XZERO + XMULT * (X data point - XOFF)
Y_n = YZERO + YMULT * (Y data point - YOFF)

For YZ format data:
X_n = XZERO + XINCR * (point number - PT.OFF)
Y_n = YZERO + YMULT * (Y data point - YOFF)
Z_n = ZZERO + ZMULT * (Z data point - ZOFF)

For XYZ format data:
X_n = XZERO + XMULT * (X data point - XOFF)
Y_n = YZERO + YMULT * (Y data point - YOFF)
Z_n = ZZERO + ZMULT * (Z data point - ZOFF)

For ENV format data:
X_n = XZERO + XINCR * (point number - PT.OFF)
$Y_{n_{max}}$ = YZERO + YMULT * (Y max point - YOFF)
$Y_{n_{min}}$ = YZERO + YMULT * (Y min point - YOFF)

Units specifications. The Units specifications include the XUNIT, YUNIT, and ZUNIT parameters. These parameters define the units (e.g., volts, seconds, etc.) for the corresponding axis. The units are specified as a literal string following the UNIT parameter (e.g., XUNIT:S, YUNIT:V).

Binary data field width (BYT/NR). This parameter specifies the number of bytes transmitted per value in the curve data. For most curve data the value is either BYT/NR:1 (one byte per value) or BYT/NR:2 (two bytes per value). This parameter is omitted when the curve is transmitted in ASCII.

Binary number format (BN.FMT). This parameter specifies the format of the binary data. There are four possibilities:

BN.FMT:LF = Left-justified binary fraction
BN.FMT:FP = Binary floating-point number
BN.FMT:RI = Binary integer
BN.FMT:RP = Right-justified binary positive integer

The instrument manuals describe the data format used by the instrument in more detail. This parameter is omitted if the curve data is transmitted in ASCII.

Binary data precision (BIT/NR). This parameter specifies the maximum number of significant bits in the curve data. Since data must be transmitted on the GPIB in eight-bit bytes, if the output data from the instrument does not contain an integer multiple of eight bits, some bits will be unused. This parameter specifies the number of bits that are significant in the output data. For eight significant bits, the parameter is BIT/NR:8. This parameter is omitted if the curve data is transmitted in ASCII.

Curve error check (CRVCHK). This parameter specifies the type of error checking used in the curve data. The only type currently used in Tektronix instruments is an 8-bit checksum. The parameter is CRVCHK:CHKSM0. Notice that binary data transmitted in the end block format does not include error checking, so the CRVCHK parameter is CRVCHK:NONE. This parameter is omitted if the curve data is transmitted in ASCII.

Byte error check (BYTCHK). This parameter specifies the type of error checking used in each byte, if any. At this time, no byte error checking is implemented in Tektronix instruments, so the parameter is either omitted or is specified as NULL. This parameter is omitted if the curve data is transmitted in ASCII.

The Curve Data

The second part of a Tektronix Standard Codes and Formats waveform is the curve data. This part contains a header word (CURVE), followed by the waveform data. A few instruments also include a curve identifier string that is similar to the waveform identifier (WFID) string in the preamble. Some instruments may omit the CURVE header word preceding a binary block transmission. The syntax of the curve data is:

CURVE [CRVID:optional i.d. string,]<curve data>

The CURVE header and, if included, curve identifier string are always sent in ASCII, regardless of the format of the curve data. Curve data may be sent in binary or ASCII. When sent in ASCII, the curve data consists of a series of ASCII numbers separated by commas. A typical curve might look like this:

CURVE 1.245,1.985,2.333,2,986,-1.256,...

The other form for curve data is called binary block. Binary block data is transmitted in the form:

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

% is the ASCII code for a "%" character. This identifies the data that follows as a binary block.

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 sends more than 8 bits of data per data point, two bytes are used per point with the most significant byte sent first.

CHECKSUM is an 8-bit binary number that is the two's complement of the modulo-256 sum of all preceding bytes except the first (%). In other words, it is the two's complement of the 8-bit sum of the preceding bytes, ignoring the carry. If the listener computes a modulo-256 sum of all the bytes except the percent sign, but including the checksum, the result should be zero.

Some Typical Waveforms

The Tektronix Standard Codes and Formats provide for a standard waveform format, but it still provides some options for adapting the standard to a particular instrument's needs. Some example waveforms from Tektronix programmable instruments are shown here as an aid to understanding the Tek Codes and Formats waveform format.

7D20 Programmable Digitizer waveforms. The Tektronix 7D20 Programmable Digitizer can send waveforms in both ASCII and binary format. It can also send the waveform data and preamble together or separately. A typical waveform in each format is shown.

The WFMPRE? query returns the preamble only. Typical preambles for a waveform sent in binary and for a waveform sent in ASCII are shown below.

Binary waveform preamble:

```
WFMPRE WFID:W 1,ENCDG:BINARY,NR.PT:1024,
PT.FMT:Y,XINCR:1.0E-5,PT.OFF:1.2E+1,XZERO:0,
XUNIT:S,YMULT:1.0,YZERO:0,YUNIT:V,BYT/NR:1,
BN.FMT:LF,BIT/NR:9,CRVCHK:CHKSUM0
```

ASCII waveform preamble:

```
WFMPRE WFID:W 1,ENCDG:ASCII,NR.PT:1024,
PT.FMT:Y,XINCR:1.0E-5,PT.OFF:1.2E+1,XZERO:0,
XUNIT:S,YMULT:1.0,YZERO:0,YUNIT:V
```

The CURVE? query returns only the CURVE header followed by the curve data in either ASCII or binary, depending on which mode is selected with the DATA command. A typical ASCII waveform, for example, is shown below:

```
CURVE 11.84,1.88,2.02,2.18,...,-3.14,-3.20
```

The WAVFRM? query returns both the preamble in the form shown above followed by the CURVE header and curve data. The preamble is separated from the curve by a semicolon. A typical response to the WAVFRM? query is shown below:

```
WFMPRE <preamble>;CURVE<waveform data>
```

7854 Programmable Oscilloscope waveforms. The 7854 Programmable Oscilloscope sends its waveforms in ASCII format only and always with the preamble. A typical 7854 waveform is illustrated below.

```
WFMPRE WFID:FULL,ENCDG:ASC,NR.PT:500,PT.FMT:Y,
PT.OFF:500,XINCR:1.0E+3,XZERO:1.0E+9,XUNIT:HZ,
YOFF:225,YMULT:4.0E-1,YZERO:0,YUNIT:DBM
```

Notice that the preamble is separated from the CURVE data by a semicolon and either a carriage return (when the 7854 message terminator is set to EOI only) or a carriage return and a line feed (when the 7854 message terminator is set for LF or EOI).

7612D Programmable Digitizer waveforms. The 7612D sends its data in binary block format with no CURVE header or preamble. The standard binary block format is used with the exception that a semicolon message unit delimiter is appended to the end of the block after the checksum. The format of the 7612D's binary block output is:

```
%<byte count><byte count><data values><checksum>;
```

492P Programmable Spectrum Analyzer waveforms. The 492P Programmable Spectrum Analyzer can send waveforms in either ASCII or binary format. The preamble and the waveform are always sent separately. The WFMPRE? command returns a preamble of the form:

Binary waveform preamble:

```
WFMPRE WFID:FULL,ENCDG:ASC,NR.PT:500,PT.FMT:Y,
PT.OFF:500,XINCR:1.0E+3,XZERO:1.0E+9,XUNIT:HZ,
YOFF:225,YMULT:4.0E-1,YZERO:0,YUNIT:DBM,
BN.FMT:RP,BYT/NR:1,BIT/NR:8,CRVCHK:CHKSM0,
BYTCHK:NULL
```

ASCII waveform preamble:

WFMPRE WFID:FULL,ENCDG:ASC,NR.PT:500,PT.FMT:Y, PT.OFF:500,XINCR:1.0E+3,XZERO:1.0E+9,XUNIT:HZ, YOFF:225,YMULT:4.0E-1,YZERO:0,YUNIT:DBM

The CURVE? query in the 492P returns the CURVE header followed by a curve identifier string and the curve data. A typical ASCII curve is:

CURVE CRVID:FULL,75,76,76,77,78,80,83,88,...

In binary format, the response to the CURVE? query is the same except that a binary block is sent in place of the ASCII curve data. The form of the response is:

CURVE CRVID:FULL,<% binary block>

abort—To terminate the execution of a program or command. When a program is aborted, it can be re-started, but it cannot be continued from the point it was aborted.

active call sequence—The sequence of program segments that are currently executing or awaiting the completion of another segment. The active call sequence always includes the currently executing program segment and, if the current segment was called by another segment, the segments that led to the call of the current segment. For example, consider a main program that calls subprogram A, which calls user-defined function B, which invokes interrupt handler C. During the execution of handler C, the active call sequence includes C, B, A, and the main program, in that order.

address—A numeric code that represents a unique location or device. In the 4041, each location in memory has a unique address. Likewise, each device on the GPIB has a unique address.

analog-to-digital conversion (A/D)—The process of converting a continuously variable voltage or current to a digital code that represents the value of the voltage or current.

AND—A Boolean algebraic operation that yields an output value of true or "1" when both input values are true or "1" and false or "0" otherwise. In 4041 BASIC, the AND operator returns "1" if both operands are true or "1." The BAND operator performs an AND operation on each bit of two values and sets the corresponding bit to "1" in the result if the bit in both values is "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 or string data items referenced by a single variable name. In 4041 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 alphanumeric characters, a standard set of punctuation characters, and a standard set of control characters (see control character).

ASK$ parameter—A parameter whose current setting can be read using an ASK$ statement.

ASK$ parameter—An I/O parameter whose current setting can be read using an ASK$ statement.

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.

ATN—The GPIB Attention line. The controller-in-charge asserts ATN when it is transferring multi-line 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.

BASIC—An acronym for Beginner's All-purpose Symbolic Instruction Code. BASIC is a simple easy-to-learn computer programming language. 4041 BASIC is a highly enhanced version of standard BASIC that provides many extensions for instrument control and programming ease.

baud—A measurement of the speed of data transfers, usually applied to serial data transfers only. Generally, a baud is equal to one bit per second and the number of characters per second is approximately the baud rate divided by 10. The baud rate for the 4041 COMM ports is set with the BAUd parameter in a SET DRIVER statement.

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. Thus, the number 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, and optionally, a terminator character. The binary block may or may not be preceded by an ASCII header word such as "CURVE."

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 4041, the I/O buffer is an area of memory where data for transmission is stored temporarily during an input or output operation. The BUFFER clause in 4041 BASIC allows the user to define a string variable for use as an I/O buffer.

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—A statement or method used to branch to or transfer control to a specified subprogram, user-defined function, or ROM routine. In 4041 BASIC, the CALL statement is used to transfer control to a subprogram. User-defined functions are invoked using a standard function syntax. ROM routines are invoked with the RCALL statement. Yet, all of these branches are generally referred to as a "call" to the routine, even though some don't use the CALL statement.

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 two's complement of the sum of all the bytes in the block, excluding the block header but including the byte count bytes. If the sum is larger than can be contained in 8 bits, it is truncated to the least significant 8-bits.

clause—A modifying word included in a 4041 BASIC I/O statement. The clause modifies the default I/O parameters. For example, the # clause specifies an alternate I/O source or destination in place of the default I/O path. The BUFFER clause defines a string variable as an alternate I/O buffer.

console device—The primary device through which the user interacts with a computer. In 4041 BASIC, the console device can be changed with the SET CONSOLE statement. The power-up default console device is the front panel.

control character—An ASCII character that doesn't represent a printing character (e.g., alphanumerics, punctuation, etc.). Instead, control characters initiate special control functions. These characters are usually sent by holding the CONTROL key on keyboard while pressing another character. Some control characters have dedicated keys. For example, the carriage return character (CONTROL-M) is a control character. Other control characters are defined to control handshaking with a device, execute a tab, execute a form feed, etc.

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.

controller-in-charge—The controller that is currently in charge of bus operation. The controller-in-charge may or may not be the system controller (see system controller). A device may become the controller-in-charge if it is capable of assuming control and the current controller-in-charge passes control to it with the TCT (Take Control) message.

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. In Tektronix instruments, DCL aborts I/O operations, clears internal I/O buffers, halts operations in progress, resets the status byte (except power-up), and clears and SRQs (except power-up). See SDC.

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.

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.

device-dependent message—A message transferred on the GPIB with ATN un-asserted. 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.

DMA—Direct Memory Access; a technique of transferring data directly from a device into a computer's memory or vice-versa without involving the CPU. This process is considerably faster than the normal method of transferring bytes one at a time using the CPU to move the data from the device into memory. The Option 1 GPIB interface on the 4041 supports DMA transfers.

driver—A subroutine or set of subroutines that controls an I/O interface in a computer. For example, the GPIB driver in the 4041 takes I/O requests from the 4041 operating system and controls the GPIB interface hardware. The driver is normally transparent to the user because the operating system passes data to the driver internally—the user seldom interacts directly with the driver program.

end block—A data format defined by Tektronix Standard Codes and Formats for transferring binary data of indefinite length. The end block starts with the "@" character followed by the binary data values. End block transmissions are always terminated with EOI.

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 also asserts EOI to indicate the end of the message. EOI is also used during the parallel poll process.

equivalent-time digitizing—A digitizer that captures samples 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 trapping—When an error occurs while a program is executing, the computer's operating system normally takes control and reports an error message to the user. Depending on the nature of the error, program execution may be terminated or may continue after the error is reported. 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.

function—A special type of program segment that returns a single numeric or string result. In 4041 BASIC there are two types of functions: pre-defined numeric and string functions such as SIN (the sine function) and SEG$ (segment of a string function), and user-defined functions (program segments that begin with a FUNCTION statement).

GET—Group Execute Trigger; an addressed GPIB interface message. The GET message is executed only when:

1. The instrument is addressed to listen
2. The DT function is implemented (DT1 subset)
3. The DT function is enabled.

When these conditions are met and GET is received, the instrument initiates an instrument-dependent function, such as an acquisition. For example, a digitizer that implements the DT1 subset can be set to begin an acquisition when it receives GET.

global variable—A variable that can be referenced anywhere in a program, in contrast to a local variable which can only be referenced within the program segment it is defined in. In 4041 BASIC, global variables may include numeric or string variables as well as line labels.

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 Tektronix 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.

IFC—The GPIB Interface Clear signal line. The system controller asserts this line to reset all interfaces to a known state. If the system controller is not the current controller-in-charge, asserting IFC also returns control to the system controller.

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 uni-line messages (a single line asserted, such as REN, SRQ, EOI, etc.) or multi-line messages (bytes sent on the data lines with ATN asserted). Multi-line interface messages may be universal messages affecting all devices on the bus (such as DCL), or addressed messages affecting only the addressed devices (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).

invoke—To put into effect or operation; to call (see call). A CALL statement invokes a subprogram.

I/O—An acronym for Input/Output.

keyword—Some words have special pre-defined meanings to a computer. These words cannot be used as variable names or line labels within a program. For example, to the 4041, the word PRINT always means "output data." 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 used by the controller-in-charge 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).

listener—A GPIB device that has been addressed to listen by the controller-in-charge 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).

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 work together on the bus without a controller.

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. In 4041 BASIC, local variables may include numeric or string variables as well as line labels.

logical parameter—A parameter that sets the characteristics of a particular device or data transfer. For example, the bus address of a GPIB device and the end-of-message character are GPIB logical parameters.

logical unit number (LUN)—A number that represents a pre-defined stream specification. The OPEN statement in 4041 BASIC associates a LUN with a stream spec. After the OPEN statement is executed, the LUN can be used in place of the full stream spec. On power-up, LUNs 0-30 are associated with GPIB devices at the corresponding address and all other LUNs refer to the console device.

module—A computer program is often divided into smaller parts, called subprograms. A single subprogram or small group of subprograms that perform a single function within a program is often referred to as a module. Good programming practice suggests that a program be written as a set of small independent modules.

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 release DAV or go on to the next handshake cycle.

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.

physical parameters—A parameter that affects all data transfers through the specified interface or driver, regardless of the LUN used. The baud rate of a COMM port is a physical parameter.

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).

proceed mode—4041 BASIC allows a program to continue processing after initiating an I/O operation before the I/O is complete. This capability is called Proceed Mode.

pseudo-random digitizing—A type of equivalent-time digitizing where the sampling rate and the input signal repetition rate are asynchronous. Thus, samples are taken at what appear to be random points on the input signal. The points are not actually random, since the sampling rate is fixed, but they appear to be random with respect to the input signal (see equivalent-time digitizing).

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 be written to as well as read. RAM is used in the 4041 for storing BASIC programs and data.

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 unit, 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 files, such as on the 4041 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 a computer file, the record length is the number of characters or bytes in each record.

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. 4041 BASIC and the ROM routines are stored in integrated circuits that are factory programmed with the operating system program. This information can only be read by the computer, it cannot be modified or written over.

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 or string value.

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.

secondary address—A numeric code 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 (see equivalent-time digitizing).

serial—Handling or transferring data one item at a time, in contrast to parallel, where several data items are handled or transferred at once. Data is transferred over the 4041 COMM ports in bit serial fashion. Each bit in a character is transferred one at a time, starting with the most significant bit and ending with the least significant bit.

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. Thus, the name serial poll.

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-in-charge. The controller usually conducts a serial poll or a parallel poll to determine which device is asserting SRQ and to read the device status.

stream specification or stream spec—4041 BASIC I/O is device-independent. In other words, in most cases, you can use the same I/O statement to transfer data to or from several different devices. The source of input data or destination for output data is defined by a stream specification (or stream spec for short). The stream spec specifies which driver will process the data (GPIB0, GPIB1, COMM0, COMM1, FRTP, etc.) as well as the parameters that control the way data is transferred.

subprogram—A program segment in 4041 BASIC that is bounded by a SUB and an END 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 and SUB statements.

subroutine—A part of a 4041 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. Subroutines may be called by line number or by a line label.

system controller—The default controller-in-charge for a GPIB system. The system controller powers up as the controller-in-charge. The system controller is the only device on the bus that can assert IFC (Interface Clear). If control has been passed to another device, asserting IFC returns control to the system controller.

system device—The device that the 4041 loads programs from and uses by default for all file I/O.

talk address—The primary address of a GPIB device plus 64. The talk address is the address used by the controller-in-charge 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).

talker—A device on the GPIB that has been addressed to talk by the controller-in-charge 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).

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.

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. It also defines a common feature set for system products that makes system programming easier.

Index

A

addressing 3, 8, 9, 11, 12, 13, 14, 29, 46, 51, 94
ASK parameters 21, 51
ATN 3, 11, 27, 30, 46

B

block binary data 35, 37, 38, 44, 45, 53, 93, 109
BUFFER clause 40, 41, 42, 43, 91, 94
byte count 35, 37, 38, 46, 115

C

checksum 35, 37, 38, 46, 115
console device 20, 46
controller 1, 4, 10, 11
controller-in-charge 10, 11, 12, 25, 46, 48, 49, 50, 61, 68

D

data logging 7, 12
debugging 2, 5, 18, 22
DEL parameter 27
DELN clause 39, 40, 44
DELS clause 39, 40, 44
Device Clear (DCL) 5, 22, 47, 57, 68
device-dependent messages 3, 13, 28, 29
Direct Memory Access (DMA) 15, 94, 95, 104, 106, 107
DM 5010 Programmable Digital Multimeter 4, 31, 86, 88, 98, 110

E

end block binary data 35, 37, 38
EOA (End of Argument) 26, 27, 28, 39, 45, 53, 95
EOH (End of Header) 26, 27, 28, 31, 45, 53, 95, 104
EOI (End or Identify) 3, 13, 14, 22, 28, 30, 38, 46, 47, 48, 49, 57, 67
EOM (End of Message) 13, 14, 28, 38, 39, 50, 53, 95, 104
EOQ (End of Query) 26, 27, 95
EOU (End of Unit) 26, 27, 28, 36, 39, 45, 53, 95, 104
error handling 2, 7, 22, 57, 58, 59, 66, 72

F

FG 5010 Programmable Function Generator 41, 49, 102

G

GETMEM 38, 41, 43, 44, 45, 94, 104
global variable 2
Go To Local (GTL) 46, 47
Group Execute Trigger (GET) 8, 46, 47

H

handler 57, 58

I

Interface Clear (IFC) 11, 22, 46, 47, 57, 68
interface messages 3, 8, 46
interface subsets 8
interrupt handling 4, 22, 57
I/O buffer 40, 41, 42, 92, 105, 106, 107, 109
IST parameter 27
item format 54

L

line labels 18
listen address 9, 30
listen only 11, 12
listener 1, 11, 30, 67
Local Lockout (LLO) 5, 47
local variable 2, 17, 18
logical mode (tape) 52
logical parameters 21, 25, 26, 27, 28
logical unit numbers (LUN) 18, 21, 25, 28, 30, 31, 41, 46, 50, 51, 55, 57, 58, 60, 61, 67, 72, 78, 81, 91, 95

M

MA parameter 14, 25, 50, 94
message terminator - See EOM
MI 5010 Programmable Multifunciton Interface 110
MLA 22, 46, 47, 49, 52, 57, 68, 69
MTA 22, 46, 47, 52, 57, 68, 69

P

physical mode (tape) 52, 53, 54, 55
physical parameters 21, 25, 26
PNS parameter 26
power-up 25
PRI parameter 27
primary address 3, 9, 27, 31, 50, 96
proceed mode 18, 40, 41, 43, 44, 72, 94, 106, 107, 111
PROMPT clause 32, 36, 41, 94
PS 5010 Programmable Power Supply 29, 31, 32, 63, 89
PUTMEM statement 45, 46, 92, 94, 95

Q

query commands 3, 27, 32, 49

R

RBYTE 49, 51, 53, 55, 98, 104, 105, 106
Remote Enable (REN) 3, 11, 46, 47

S

SC parameter 26, 49
SEC parameter 27
secondary address 3, 9, 27, 30, 31, 32, 96
SELECT statement 46, 58, 61, 62
Selected Device Clear (SDC) 46, 47, 68
serial poll 48, 96, 97, 98
SI 5010 Programmable Scanner 110

SPE parameter 28, 62, 98
SRQ interrupts 22, 25, 34, 48, 57, 58, 59, 60, 61, 62, 67, 69, 96, 104
status byte 4, 25, 62, 63, 65, 96
stream specifications 20, 21
subprogram 2, 5, 17, 59, 70, 71
system Controller 10, 11, 26, 46, 49
system device 20

T

Take Control (TCT) 48, 50, 57, 70
talk address 9, 30, 32
talk only 11, 12
talker 1, 11, 29, 67
Tektronix Standard Codes and Formats 5, 13, 18, 27, 31, 34, 64, 66
TC parameter 27
TL parameter 51
TIM parameter 25, 28, 51
TRA parameter 28

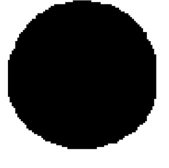
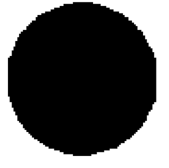
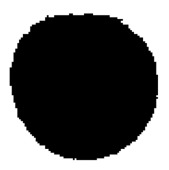
U

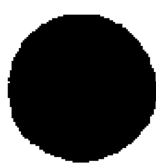
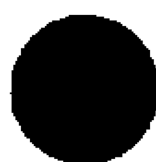
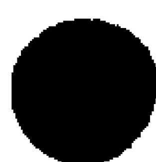
Unlisten (UNL) 9, 30, 32, 46, 48, 49, 94, 95, 96
Untalk (UNT) 9, 30, 32, 46, 48, 49, 95, 96
user-defined function 17
user-defined keys 2, 7, 22, 73
using clause 36, 37, 38, 82, 91, 93, 96, 104, 105

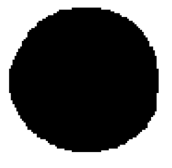
W

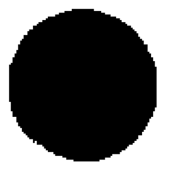
waveform data 3, 4, 7, 34, 35, 75, 89, 109, 113, 115
WBYTE 46, 47, 48, 51, 53, 55, 68, 69, 98, 104, 105

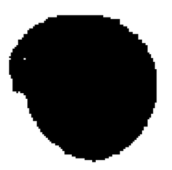
390AD Programmable Digitizer 103
492P Programmable Spectrum Analyzer 38, 115
1360P Programmable Switcher 110
5223 Digital Storage Oscilloscope 35, 36
7D20 Programmable Digitizer 34, 37, 42, 43, 90, 95, 102, 106, 109, 115
7S12 TDR/Sampler 103
7612D Programmable Digitizer 3, 9, 14, 30, 31, 34, 44, 67, 81, 85, 86, 103, 115
7854 Waveform Processing Oscilloscope 4, 39, 88, 98, 103, 106, 110, 115







4041 GPIB Programming Guide