TEK SPS BASIC V02/V02XM System Software

CP57000/CP57500

Tektronix®

COMMITTED TO EXCELLENCE

Tektronix®
COMMITTED TO EXCELLENCE

TEK SPS BASIC V02/V02XM System Software

CP57000/CP57500

INSTRUCTION MANUAL

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

Serial Number ____________________

First Printing JAN 1980 PRODUCED BY SPS DOCUMENTATION GROUP 070-2501-00

PREFACE

This manual describes the commands, the functions, and the low-level IEEE 488 Interface driver that are included in the TEK SPS BASIC VØ2 and VØ2XM System Software package. The peripheral device drivers that are a part of this package are discussed in a separate manual. Other TEK SPS BASIC VØ2/VØ2XM packages are described in additional manuals which you should consult when these software packages are added to your system.

All releases of TEK SPS BASIC VØ2/VØ2XM System Software are documented by this manual. Any exceptions to a command, function, option, or capability being supported by a specific release of the software are noted where appropriate. Information that pertains only to extended memory (XM) systems is shaded.

This is not a BASIC primer. It is assumed that you are familiar with a high-level language and need this manual as a reference for TEK SPS BASIC and its signal processing and waveform handling techniques. It is recommended that you read the first three sections before you start to program, and then consult the remainder of the manual as needed. An attempt was made to make the information more retrievable by using topical footers in sections 4, 5, 6, and 8 and by printing the error codes on colored paper. If you have comments on what was done or suggestions for how to improve future manuals, please write via the "Your Comments Count" form in the back of the manual.

TABLE OF CONTENTS

@

@

SELECTION LOCATION GUIDE

SECTION 1

INTRODUCTION

TEK SPS BASIC VØ2 is a powerful general-purpose programming language which offers sophisticated tools for the control of instruments plus the acquisition, processing, storage, and display of data. At the same time, it retains the easy-to-use, easy-to-learn, user-orientation of traditional BASICs. This manual describes the features that make TEK SPS BASIC VØ2 unique as well as some of the fundamentals of BASIC programming.

Main Features of TEK SPS BASIC VØ2

Modular, space-efficient and versatile, TEK SPS BASIC can serve needs ranging from a new user's five-line arithmetic calculation program to an experienced assembly-level programmer's specialized device driver or instrument control system. TEK SPS BASIC is file-compatible with Digital Equipment Corporations RT-11 software. Running on DEC's PDP-11 family of minicomputers and the TEKTRONIX CP4165 controller, it gives the programmer access, through BASIC, to features of the operating system usually accessible only through assembly-level interaction.

Here are some of the features of TEK SPS BASIC VØ2 that make this language superior to other versions of BASIC:

Choice of standard memory or extended memory versions. For those who need more than the standard 28K words of memory, TEK SPS BASIC VØ2 is available in an extended memory (XM) version as well as a standard memory version. Intended for use in systems with the DEC KT11D Memory Management hardware, the XM version allows up to 96K words of extended memory to be used for numeric array storage, for a total of 124K words of memory. Yet, except for timing considerations, most of the differences between the standard and XM versions are transparent. The full complement of TEK SPS BASIC software packages is obtainable in either the standard or XM version.

Adaptable and expandable systems. The modular construction of TEK SPS BASIC gives you the freedom to design a processing system that fits your needs, rather than a general need. A variety of TEK SPS BASIC VØ2 software packages can be used with the system software. In addition, by using optional packages you can write your own commands for special applications. Likewise, you can write software drivers for one-of-a-kind instruments.

Full graphics capability. A graphics package is available to help you display your results in an easy-to-understand format. These graphics tools range from powerful commands like GRAPH and DISPLAY, which accept arrays and waveforms as arguments and draw the data in them with axes and graticules, to a complete selection of move and draw commands which permit the user to specify destination locations as absolute or relative, in user or screen units.

Signal analysis capability. A set of signal analysis commands is also available that greatly simplify computations such as Fourier transforms, convolution, correlation, differentiation, and integration. Only one BASIC statement is needed to perform each of these tasks.

Flexible instrument control. Drivers and/or command packages are available for Signal Processing Systems instruments such as the TEKTRONIX Digitizing Oscilloscope (DPO), R7912 Transient Digitizer, 7912AD Programmable Digitizer, and the 7612D Programmable Digitizer. The system is also designed to support and control general-purpose instrumentation. User-written drivers and commands are easy to integrate into TEK SPS BASIC and can be created by someone competent in assembly language programming, using an optional command package. Instrument communication commands are clear and general, efficient at moving data to and from instruments, and sufficiently flexible to permit control of instrument idiosyncrasies in communication.

GPIB Drivers. For control of devices connected to a General Purpose Interface Bus (GPIB), TEK SPS BASIC VØ2 includes a low-level IEEE 488 Interface driver. This driver gives you control over either a CP4100/IEEE 488 (Q-bus) Interface or a CP1100/IEEE 488 (UNIBUS^{R}) Interface. Commands range from primitive, line-level controls which accept string arguments describing interface status to higher level directives which make GPIB housekeeping details transparent to user programs. Interface time-out duration is under user control, instrument data can be acquired using direct memory access (DMA) protocol, and the driver recognizes error, EOI, and SRQ conditions directly.

A second GPIB driver, the high-level IEEE 488 Instrument driver, is available as a separate package. Designed especially for communication with GPIB devices that conform to the Tektronix codes and formats, it offers easier control of instruments such as the TEKTRONIX 7912AD Programmable Digitizer and the TEKTRONIX 7612D Programmable Digitizer.

Scheduling and error-handling capabilities. Designed for optimum instrument control, TEK SPS BASIC gives you complete control of the system's response to errors or interrupts, permitting the scheduling of routines based on real-time instrument events, relative or absolute time delays, or error conditions at any of 127 software-assigned priorities. Instrument control functions can be segregated into separate tasks, protecting individual tasks from being halted by errors or instrument failures in other tasks. This, plus the ability to respond to an instrument when data is available and to schedule programs, gives BASIC the power usually reserved for a time-sharing system.

Maximum control of memory space. TEK SPS BASIC gives the user maximum control over the controller memory space. Commands, drivers, and arrays can be deleted to gain memory during program execution, and initial system configuration parameters such as the number of instruments and peripherals to be supported and the inclusion of graphics and string handling capabilities are under user control. Furthermore, programs can be broken into segments which can be loaded and executed as they are needed and then deleted or overlaid when done. Even a fast overlay command, OVLOAD, is provided to speed the execution of overlaid programs.

Ease and accuracy of arithmetic operations. As a programming language at the BASIC level, TEK SPS BASIC gives you a complete software toolbox for numeric and string computations. It is accurate to 7.2 decimal digits and can operate either with or without floating-point hardware, configuring itself to produce the most space- and time-efficient calculation. TEK SPS BASIC's eight numeric and six array functions perform frequently needed operations beyond the standard five operations of addition, subtraction, multiplication, division, and exponentiation. All numeric operators and functions accept as arguments virtually any combination of numeric variables, arrays, segments of arrays, and waveforms. (A waveform is a special data structure which simplifies signal processing applications by associating a data sampling interval, a time-units label, and an amplitude-units label with an array of signal-descriptive data.)

Array, waveform, and subarray processing. A single BASIC statement is all that is needed to manipulate entire arrays or waveforms. Subarrays allow you to process only that part of an array you are interested in. Also, parts of arrays can be extracted from larger arrays or waveforms and operated on independently.

String processing. String processing, or character manipulation, is necessary to format output in a clear and easy-to-read manner. Strings in TEK SPS BASIC can be arbitrarily long and need not have their length explicitly declared. String variables are supported by a concatenation operator and nine string functions which include conversions between numeric and ASCII values of number strings. String arrays, which can be one- or two-dimensional, can easily be searched (using the MATCH command) and may have elements of arbitrary and varied lengths.

Peripheral Communication. TEK SPS BASIC communicates easily and speedily with a complete range of peripherals which include graphic terminal/keyboard, hard disk, floppy disk, cassette tape, magnetic tape, paper tape reader and punch, and line printer. Its peripheral interfacing code, like its instrument interfacing code, is written to be completely independent of the device with which it communicates. Thus, new peripheral drivers can be added to the system by TEK SPS BASIC users.

Random access files. Files are accessible and manipulable at any level from byte to block, can be read or written in ASCII or binary, can be formatted or unformatted, and can be accessed either serially or randomly.

Live keyboard. Immediate mode commands or program lines can be entered from the keyboard while a program is executing. This feature can be disabled by the LOCKKB command.

Unsolicited input. The user can interact with running programs through either prompted or unsolicited input from the keyboard.

Patching capability. Special BASIC routines are included with your software which allow you to update or correct errors in Resident BASIC or nonresident commands.

Operating Concepts

Now, let's briefly discuss some of the operating concepts of TEK SPS BASIC. Material that the beginning user may pass over is set off in square brackets.

The Interpreter

BASIC is an interpretive language. That is, as you type a program line on the terminal, the line is being stored in the controller memory. Each line is stored in sequence by line number. It doesn't matter in what order you enter program lines; the interpreter/operating system always keeps lines of text in line-number order.

When you execute a program, the interpreter/operating system takes one line at a time from memory, in line number order, and executes the command(s) in that line.

Interpretive languages are a great advantage when you are developing programs. If a mistake is found, all you need do is retype the line in error. New lines can be added at any time, and lines no longer needed can be deleted in a snap.

When you have a complete, working program it can be saved on a peripheral storage device, such as a disk or magnetic tape. These saved programs can be loaded and run again at any time.

Two Modes of Operation

TEK SPS BASIC commands can execute in either immediate mode or program mode.

Immediate Mode Operation. The BASIC interpreter can execute single commands immediately, without the command being a part of a program. When a command is typed at the terminal with no preceding line number, the command is executed immediately. It is not stored in the controller memory. Any BASIC statement can be executed in immediate mode, but some, like PRIORITY or RETURN, have no meaning in this case.

Program Mode Operation. When a BASIC statement is preceded by a line number, the statement is not executed immediately. Instead, it is stored in memory and executed when you execute the program or subroutine of which it is part. RUN, which is used to start program execution, is the only command that can not be entered in program mode.

Modules and How They Are Used

A key feature of TEK SPS BASIC is its modularity. The heart of the TEK SPS BASIC interpreter/operating system is called the Monitor/Interpreter or "Resident" BASIC. This is the software that keeps track of your program and allows you to add or delete other modules as they are used. These modules include peripheral drivers (routines that "talk" to the peripherals), instrument drivers, mathematical modules (such as the Fourier transform, correlation, differentiation, etc.), and various BASIC commands. The modular concept frees the greatest amount of memory in the controller for programs and data storage. Modularity also increases the flexibility of BASIC by allowing you to add new commands and drivers when necessary. You can even write your own.

Figure 1-1 is a block diagram of the TEK SPS BASIC VØ2 operating system. Most of Resident BASIC, which includes the system device driver, is always in memory. [A portion of Resident Basic is an overlay area into which one of two files that are stored on the system device is automatically overlaid as it is needed. The two files are TRAN.OVL and UNTRAN.OVL. TRAN.OVL translates BASIC text into the internal, executable form in which a program is stored in controller memory. UNTRAN.OVL converts the internal form back to the BASIC language form that can be displayed with the LIST command.]

Modules usually reside on the system storage device. They can be copied to other peripheral devices if necessary. The system storage device is determined when TEK SPS BASIC is initially brought into memory. It is the peripheral device from which BASIC is loaded. For most operations involving data transfers between the controller and a peripheral, the system device is assumed if no other device is specified.

Modules can be loaded in two ways. One method is with the LOAD command. When you LOAD a module, it becomes part of Resident BASIC. It is locked in and can't be released until you specifically free it with the RELEASE command.

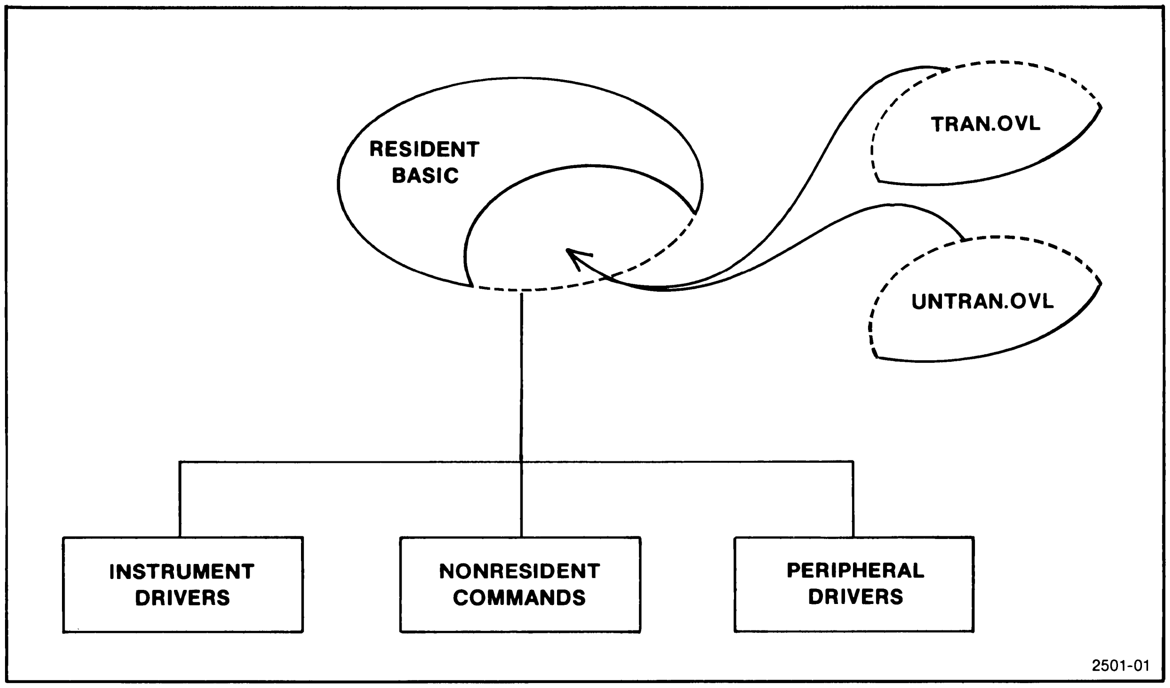


Fig. 1-1. Simplified diagram of TEK SPS BASIC VØ2.

The other method of loading modules is to let BASIC do it for you. This "auto-loading" works only with nonresident commands (not drivers) that are stored on the system device. When BASIC encounters a nonresident command in a program (or in immediate mode) and that command is not in memory, BASIC automatically fetches the nonresident command from the system storage device and continues program execution.

When a module is auto-loaded, it is not locked in. It is automatically released if room is needed later for another nonresident module, data, or program text. [A tally is kept as to how often these auto-loaded commands are used. When one has to be auto-released, the one that has had the least recent use gets released.] One of the nonresident commands, STATUS, tells you which modules are currently in memory. It also lets you know if they have been auto-loaded.

[Because of the system overlay files, care must be used if BASIC is to be executed while the system disk is removed from the system drive. Before the system device is removed, all the commands and drivers that will be needed must be in memory, none of the commands may require UNTRAN.OVL to execute, and TRAN.OVL must be resident. (TRAN.OVL can be made resident by entering an immediate mode command or by just hitting the RETURN key.)]

Figure 1-2 shows a more detailed diagram of TEK SPS BASIC VØ2. Here, individual elements from acquisition instruments to storage peripherals are presented. In the center of the diagram is the Monitor/Interpreter, or Resident BASIC. Next to it are the nonresident commands and the peripheral and instrument drivers that are loaded as needed. Resident BASIC communicates with the various peripheral devices and instruments through the drivers.

The Scheduler

As a program executes, Resident BASIC must manage its resources such as memory space and processing time in response to varying priorities, task numbers, interrupt conditions, scheduled events, nested subroutines, and other program operations. The mechanism which performs this resource management is the Scheduler.

NOTE

> An understanding of how the TEK SPS BASIC Scheduler operates is essential for the skillful and complete use of the language. However, a beginning user may want to return to this material as it becomes relevant.

The **Scheduler** consists of three structures and the routines to manipulate them. The structures are the current-job slot, the queue, and the stack (see Fig. 1-3). These contain entries, called packets, which represent the varip
of the next line to be executed in the routine, and the internal address of the routine. The Scheduler always has two special packets, called idle packets, which keep the Scheduler functioning even when there are no other packets in the Scheduler. An idle packet has a priority of -1, a task number of -1, and a line number of Ø.

The **current-job slot** holds the packet of the currently executing routine, which is called the current job. When no routine is being executed (e.g., when READY has been printed on the terminal), an idle packet occupies the current-job slot. It causes Resident BASIC to loop until an entry is made into the queue. This is called idle mode.

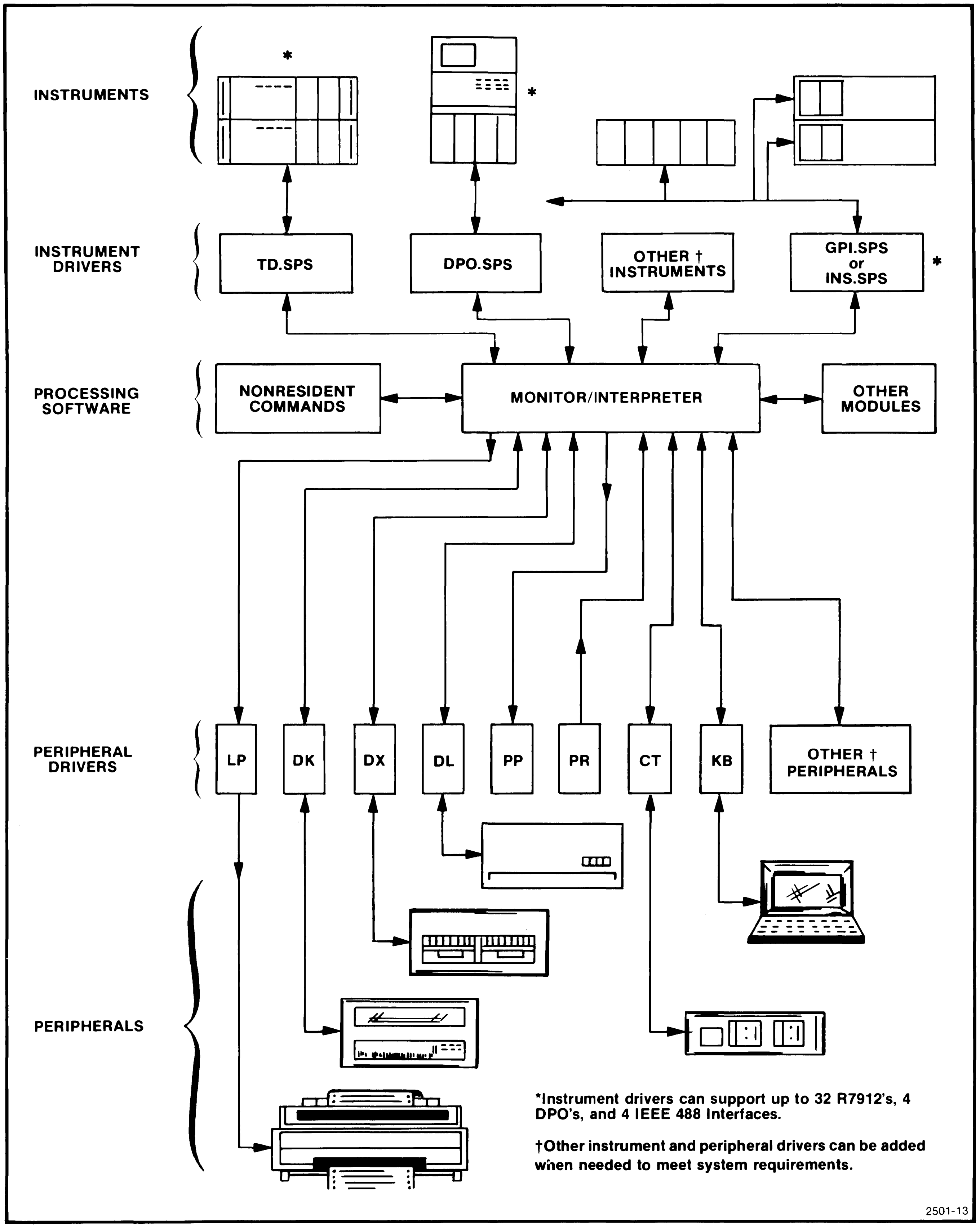


Fig. 1-2. Block diagram of the TEK SPS BASIC VØ2 Operating System.

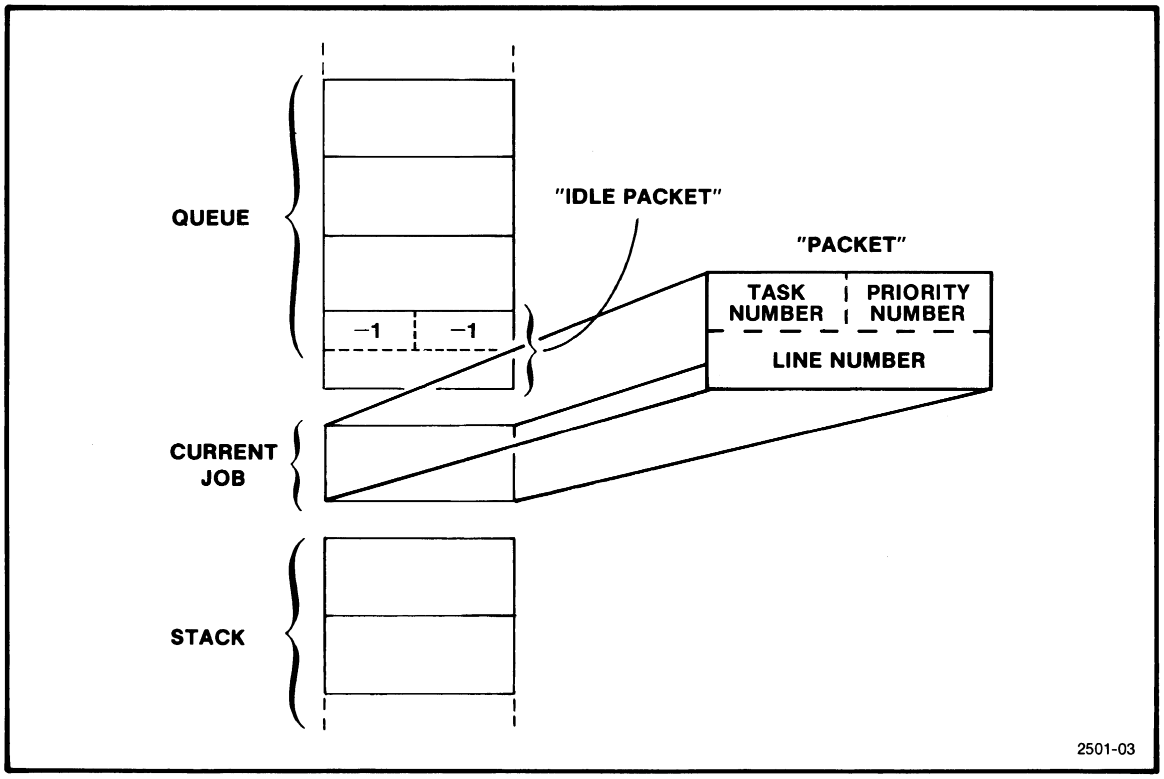


Fig. 1-3. The three structures of the Scheduler.

The **queue** contains packets for routines which are ready to receive system attention and resources. Its packets are ordered by priority. The highest priority routine which has not yet begun execution is at the top of the queue. The queue always holds an idle packet. When the queue is "empty", the idle packet is at the top of the queue.

The **stack** contains packets for routines which have begun to execute but which were "bumped" from the current job by a higher priority routine. When the idle packet is bumped from the current-job slot (BASIC is no longer in idle mode), the idle packet is put onto the stack.

Packets enter the Scheduler through the queue. Packets can be put into the queue in several ways. An immediate mode statement causes packets to be entered that edit, translate and execute the statement. A WHEN statement can make the system respond to an interrupt by scheduling an interrupt-handling routine which is represented by a packet inserted into the queue. A SCHEDULE statement can cause a particular routine to be scheduled for execution at a particular time; when that time occurs, a packet describing that routine is entered into the queue.

The Scheduler manipulates the packets in its three structures using the priority and task number assigned to each routine as it is entered.

Priority is like a ranking of routines by importance. A higher number means a particular routine is more important and should be done sooner. Priorities range from Ø to 127, with 5Ø the default. They can be assigned by using the PRIORITY command or the WITH keyword in the WHEN or SCHEDULE commands. Packets for immediate mode commands are entered into the queue with a priority of 127.

The **task number** is a numeric name associated with a routine when its packet is entered into the queue. Unlike priority, it implies no order of importance, but is used only to identify different parts of the same task. Task numbers between Ø and 126 are assigned by using the AS TASK keywords in the RUN, WHEN, and SCHEDULE commands. If AS TASK is not specified, WHEN or SCHEDULE assigns the task number associated with the WHEN or SCHEDULE command as it executes. Immediate mode commands are put into the queue with a task number of 127.

The Scheduler uses the priority to monitor the execution of routines and to determine whether a routine should continue executing or be suspended to allow another routine to begin executing. At the completion of the execution of each command of the current job, the Scheduler compares the priority of the current-job packet against the priority of the packet at the top of the queue. If the current job has a higher or equal priority, its next command executes. If not, the packet in the current-job slot is pushed onto the top of the stack, and the packet at the top of the queue is moved to the current-job. The first command of the new current job is executed.

If there is no next command in the current job (e.g., RETURN has just been executed), the Scheduler compares the priority of the top packet on the stack with the priority of the top packet on the queue. Whichever packet has the higher priority is moved to the current-job slot and the first command of the routine it represents executes. If the packets are of equal priority, the packet from the stack is selected. (See Fig. 1-4.)

The task number is used to limit the impact of fatal errors. If a fatal error occurs and the user has not set up other error-handling

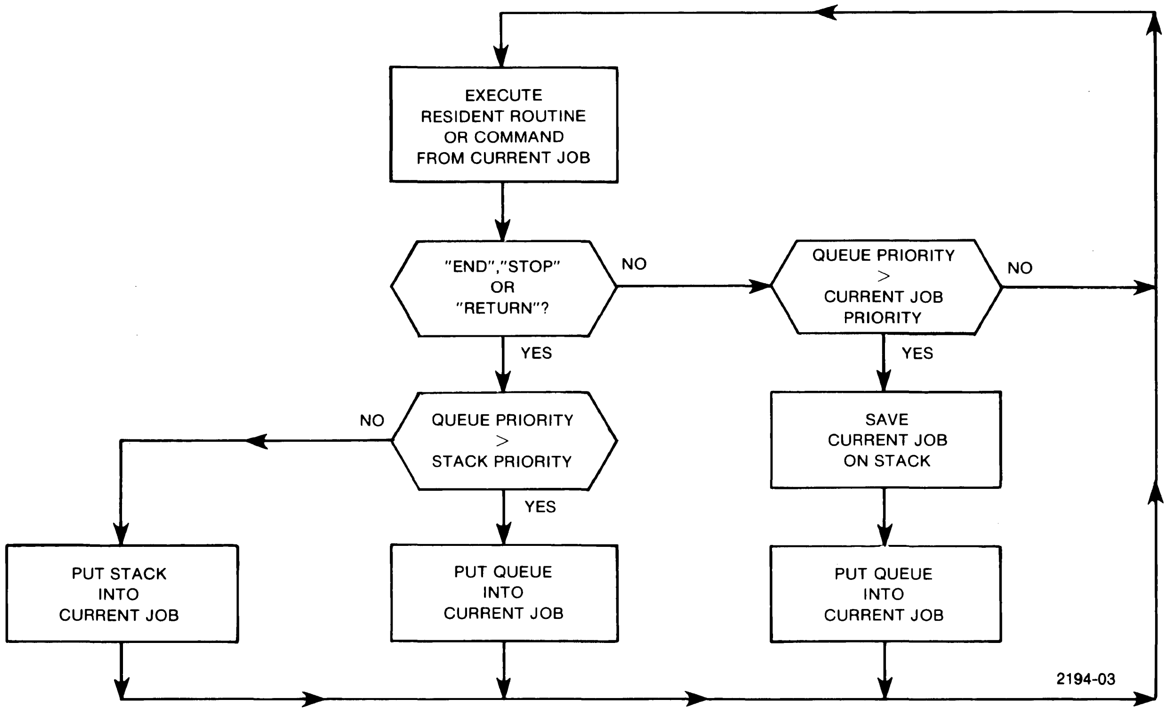

EXECUTE RESIDENT ROUTINE OR COMMAND FROM CURRENT JOB
"END","STOP" OR "RETURN"?
NO
YES
QUEUE PRIORITY > CURRENT JOB PRIORITY
NO
YES
QUEUE PRIORITY > STACK PRIORITY
NO
YES
SAVE CURRENT JOB ON STACK
PUT STACK INTO CURRENT JOB
PUT QUEUE INTO CURRENT JOB
PUT QUEUE INTO CURRENT JOB
2194-03

Fig. 1-4. Diagram of the Scheduler's priority-based execution process.

procedures with the ONERR command, the Scheduler removes from its current-job slot, queue, and stack all packets with the same task number as the task generating the error. Packets with a different task number remain. This gives the user the capability of confining the impact of errors to those program segments which will be affected and permitting other program segments to continue to execute. (See Section 8 for a complete discussion of Resident BASIC's default error-handling procedures. Also, for a discussion of the user's error-handling options, see the ONERR command in Section 4.)

Defining the Terms

Before going further into TEK SPS BASIC programming, it is necessary to define some terms used throughout this manual. These terms describe the fundamental parts of BASIC and waveform processing. Further definitions can be found in the Glossary section of this manual.

Program Structure and Control

Program. A BASIC program is a set of one or more numbered statements (instructions) that performs some operation when executed. A program can be made up of one or more parts called subprograms or subroutines.

Statement. A statement is an instruction to the software to take some action. For example, the statement LET X = 3 tells the software to place the value of three into the variable named X. The statement name is LET. Another statement is PRINT X. It tells the software to print the value of X on the computer terminal. In TEK SPS BASIC, statements are also called commands, and the two words can be used interchangeably.

Subprogram. A subset of a program is called a subprogram or routine. It is a group of statements within a program that does a particular job.

Subroutine. A subroutine is a subprogram that terminates with a RETURN statement. It can be repeatedly called and/or it can be called from different parts of a program.

Line Number. Line numbers are used to order the execution of statements in a program. Line numbers are integers between 1 and 32767, inclusive. Statements with line numbers can be entered in any order, but they are always stored and listed in ascending order. Statements are always executed in line number order unless a GOTO or GOSUB is performed or a higher priority routine interrupts.

Program Mode. This is one of the two modes under which a statement can execute. A program-mode statement is preceded by a line number, and each such statement is stored in the controller memory as it is entered. Program-mode statements are not executed until the program execution is begun by a RUN, OLD, CHAIN, GOTO, or SCHEDULE command, or by an interrupt-driven start-up.

Immediate Mode. This is the second mode of statement execution. An immediate mode statement has no line number and is not stored. An immediate mode command is scheduled for execution as soon as it is entered from the terminal.

Task. A task is a subprogram distinguishable from another subprogram by the task number associated with it. A fatal error can occur in one task and halt its execution without halting other tasks.

Task Number. This is the numeric name assigned, either explicitly or by default, to a routine when it is scheduled for execution (entered in the Scheduler queue). The name implies no order of importance or priority of execution. It is only a way of identifying different subprograms in the Scheduler. A task number between Ø and 126 can be explicitly assigned by RUN, WHEN, or SCHEDULE. Immediate-mode statements always execute as task number 127.

Priority. Priority is the rating of the relative importance of a task. It is the basis for deciding in what order tasks execute.

Priority Number. The priority number assigned to a routine determines in what order it will execute in relation to other routines. Also, a routine with a high priority number can interrupt a routine with a low priority number. Programs RUN with a default priority of 5Ø, but the PRIORITY command can change the priority number of a currently executing routine. The WHEN or SCHEDULE command can assign a priority number from Ø to 126 to a routine. Immediate-mode statements execute with a priority number of 127.

Special Characters

Backslash Character. The backslash character (\) is used to define the end of a command when more than one command appears on a single line. This character is entered on some terminals by typing shift-L. Several commands may be entered on one line. For example:

```
100 PRINT "PROGRAM RUNNING"\GOTO 355
```

While the use of the backslash may save space when you have a series of short LET statements, overuse of the backslash can make your programs hard to read and difficult to debug.

Control Characters. The TEK SPS BASIC control characters are instructions to BASIC or to the keyboard terminal driver. Control characters are entered by holding down the CTRL key on the terminal and striking the desired character. BASIC prints an up arrow (^) followed by the corresponding character when a control character is entered.

Control-O. A Control-O inhibits the display of most output directed to the terminal. Error messages are not suppressed. A second Control-O allows the display of output. Any output sent to the terminal between the suspension and resumption of the display is lost.

Control-P. This character terminates a program and returns BASIC to the idle mode. It may be entered at any time. Control-P leaves files open, disables all WHEN interrupts, and removes any pending tasks from the Scheduler. During most operations, the Control-P is not recognized until the operation is complete. This is to insure that the data is not left in a "half-done" state.

Control-Q. A Control-Q resumes the terminal output that was temporarily suspended by a Control-S. (Control-Q is not supported by TEK SPS BASIC VØ2-Ø1.)

Control-S. This temporarily suspends terminal output until a Control-Q is entered. (Control-S is not supported by TEK SPS BASIC VØ2-Ø1.)

Control-U. This control character deletes the line being typed. All characters back to, but not including the last carriage return are deleted. BASIC responds to the Control-U by sending a carriage return and a line feed to the terminal.

Control-Z. Control-Z is used to terminate input to the COPY command when the source device is the system terminal.

Elements of Expressions

Expressions. An expression is defined by what it evaluates to. There are four types of expressions in TEK SPS BASIC: numeric expressions, array expressions, waveform expressions, and string expressions.

Numeric Expression. A constant, a variable, or any legal combination of constants, variables, waveforms, arrays, arithmetic operators, functions, and parentheses that **evaluates to a single numeric value** is considered a numeric expression.

EXAMPLE: C*SQR(2)   (C is a simple numeric variable)

NOTE

On the following pages, numeric expressions are referred to simply as "expressions."

Array Expression. An array, an array zone, or any legal combination of arrays, array zones, waveforms, numeric expressions, functions, arithmetic operators, and parentheses that **evaluates to an array** is considered an array expression. All arrays and arrays associated with specified waveforms involved in an array expression must be the same size.

EXAMPLE: 4*A2/5.6+A1   (A1, A2 are arrays of the same size)

Waveform Expression. A waveform or any legal combination of waveforms, arrays, numeric expressions, functions, arithmetic operators, and parentheses that **evaluates to a waveform** is considered a waveform expression. Within a waveform expression, all arrays and arrays associated with specified waveforms must be the same size.

EXAMPLE: W1+3*W2   (W1, W2 are waveforms with associated arrays of the same size)

String Expression. Any string constant, string variable, or any legal combination of string constants, string variables, string functions that return a string, parentheses, and the string operator (&) that **results in a string** is considered a string expression.

EXAMPLE: "THIS"&C4$ (C4$ is a simple string variable)

Numeric Constant. A numeric constant (or simply a **constant**) is a number expressed as a decimal value. A constant is entered as one or more ASCII numeric characters (0-9) with an optional decimal point and an optional positive or negative power of ten specification (E notation).

Negative constants are preceded by a minus sign; positive constants may optionally be preceded by a plus sign. Examples of legal constants are:

```
232     +33.34     6423E+5     -.ØØ5     1ØE-13
```

The E notation in a constant refers to a power of ten. In the third example, the value expressed is 6423 times 10^5 or 6423ØØØØØ.

The values expressible as a constant range from +1.7Ø141E+38 to -1.7Ø141E+38 with the smallest possible fraction expressible as ±2.93874E-39.

String Constant. A string constant (also called a string literal) is one or more characters enclosed in single or double quotes. The maximum length of a string constant is somewhat less than 80 characters (the maximum length of a line BASIC will accept minus the other characters needed in the statement). Three control characters, Control-P (returns BASIC to idle mode), Control-U (deletes line being typed), and Control-M (carriage return) cannot be included in a string literal. Some examples of string constants are:

```
"TWENTY SIX LIONS SLEPT TODAY."
"'STRINGS MAY HAVE EMBEDDED QUOTES,'HE SAID."
```

Numeric Variable. Symbols used to represent single numeric values are called numeric variables. These symbols can be one or two characters in length. The first character must be an upper case letter (A-Z). The optional second character may be either an upper case letter or a digit (Ø-9). Examples are A, Z1, and JJ. Six variable names are not allowed, as they are used as parts of statements. These are AS, AT, IF, IS, OF, and TO. A numeric variable not dimensioned to an array is called a **simple numeric variable.** An element of an array is called a **subscripted numeric variable.**

NOTE

On the following pages, numeric variables are referred to simply as "variables."

String Variable. String variables are symbols used to represent strings. The rules for naming string variables are identical to those for numeric variables, with the addition of a dollar sign (**$**). Some examples of string variable names are A$, E5$, K9$, ZZ$. The strings represented by string variables can be of any length, limited only by the amount of controller memory available. A string variable not dimensioned to an array is called a **simple string variable.** An element of a string array is called a **subscripted string variable.**

Arrays. Arrays are variables that represent more than one numeric value or string. Arrays are defined in DIM, INTEGER, or WAVEFORM statements, or they can be automatically defined in a program if certain commands are used. In standard memory systems, array length (the number of elements an array can have) is limited only by the amount of controller memory available. In extended memory (XM) systems, the maximum size of a floating-point array is limited to 8192 (8K) elements and an integer array is limited to 16384 (16K) elements. Arrays can be one- or two-dimensional. Two-dimensional arrays can be thought of as having rows and columns, or as a matrix.

Floating-Point Arrays. These arrays are defined either by a DIM or WAVEFORM statement or automatically by some commands. Names for floating-point arrays follow the same rules as names for simple numeric variables. Each element in a floating-point array requires two words of controller memory. Values in a floating-point array can range from -1.7Ø141E+38 to +1.7Ø141E+38 with the smallest possible expressible fraction being ±2.93874E-39.

Integer Arrays. Integer arrays are defined by the INTEGER statement, or automatically defined in a program if certain commands are used. Rules for naming integer arrays are the same as for naming simple numeric variables. Values in an integer array are limited to integers in the range of -32768 to +32767. Floating-point values stored in an integer array are truncated to integers. (The number 3.9 is stored as 3; a -9.9 is stored as -9.) Each element in an integer array requires one word of controller memory.

String Arrays. String arrays are defined in a DIM statement only. Each element in a string array is a complete string. The string elements can be of differing lengths. The length of any element in the string array is limited only by the amount of memory available in the controller. Names for string arrays follow the same rules as for naming string variables.

NOTE

On the following pages the term 'array' refers to floating-point or integer arrays only, not string arrays.

Array Zone. An array zone is a subarray of either a floating-point or integer array. It is a contiguous portion of an array that is accessed independently of the rest of the array. Array zones may appear anywhere that arrays may appear unless otherwise stated in the command description. They can be used with either integer or floating-point arrays, but not with string arrays or waveforms. A colon (:) between two subscripts is used to specify an array zone. For example, A(Ø:9) specifies the first ten elements of array A as an array zone.

In doubly-dimensioned arrays, one dimension may be zoned, but not both. Legal uses of array zones are shown below:

1. array(n:m,e)
2. array(e,n:m)
3. array(n:m)

Here e, n, and m are all numeric expressions yielding single values that are rounded to integers. Expression n is the subscript for the start of the zone, m is the subscript for the end of the zone, and e is a regular array subscript.

The subarray referenced in (1) above includes the elements
array(n,e), array(n+1,e),... array(m,e)

The subarray referenced in (2) includes the elements
array(e,n), array(e,n+1),... array(e,m)

The subarray in (3) includes the elements
array(n), array(n+1),... array(m)

As an example of using an array zone, let's assume you want to find the mean of a part of an array. The statement:

```
X = MEA(A1(5Ø:99))
```

would return in variable X the mean of the 5Ø elements of array A1 starting at the 51st element, A1(5Ø).

Waveforms. A waveform is a variable that represents a floating-point or integer array, a simple numeric variable, and two string variables that have been associated by a WAVEFORM statement. The array contains the data points of the waveform. The numeric variable contains the data sampling interval (DSI) which is the time between elements. The strings contain the horizontal and vertical units ('seconds' and 'volts' for example). The sampling interval and units information is automatically updated if the waveform is altered during program execution.

Operators. Operators determine the type of action to be performed on one or more quantities. Three types of operators are available in TEK SPS BASIC: arithmetic operators, relational operators, and the string operator.

Arithmetic Operators. The arithmetic operators are:

^	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

These operators are used to create an arithmetic expression.

Relational Operators. Relational operators are used in IF statements to express conditions which can be true or false. The relational operators are:

=	equal to
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
<>	not equal

String Operator. There is one string operator, the ampersand (&). It is used to make a new string by joining (concatenating) two or more separate strings. For example, the following statement creates string C$ by assigning it the result of concatenating strings A$ and B$:

```
C$=A$&B$
```

Functions. In mathematics, a function defines the value of a dependent variable based on the value of the independent variable. Similarly, a BASIC function returns a value (or in some cases, an array of values) that results from the action of the function using the given argument. A function does not change the value of this argument.

A function is not like a command. It can only be used as a part of an expression within a statement (such as LET or PRINT). Three different function types are available in TEK SPS BASIC:. numeric functions, array functions, and string functions.

Numeric Functions. Functions that operate on numeric information and return numeric information are called numeric functions. The argument of a function may be a numeric expression, an array expression, or a waveform expression. If the function's argument is a single-valued expression or constant, the function returns a single value. If the argument is an array (or waveform), the function computes a value for each element in the array (or the waveform's array), and returns an array. The numeric functions are ABS, ATN, COS, EXP, ITP, LOG, RND, SGN, SIN, SQR, and TSK.

Array Functions. Array functions always expect arrays or waveforms as the argument of the function, and always return a single value. The array functions are CRS, MAX, MEA, MIN, RMS, and SIZ.

String Functions. Functions that operate on strings or string expressions, or create strings, are called string functions. Some string functions return a number, some return an ASCII string. String-function capability may be deleted at system load time, if desired. The string functions are ASC, CAN, CHR, LEN, POS, SEG, STR, TRM, and VAL.

Instruments and Peripherals

Device Names. A device name is a two or three letter mnemonic that is used when referencing peripheral or instrument devices. The mnemonic may be followed by a device number that represents which unit of the device group is being specified. For an instrument, the device number is called a hardware unit number (HUN). For peripherals, the device number is called the drive number. In the command syntax the device name and the device number are terminated by a colon. If the device number is not included with the device name, zero is assumed. For example:

DK:	disk drive zero
DK1:	disk drive one

The following device names are currently recognized by TEK SPS BASIC:

Name:	**Device:**
KB	system keyboard terminal
DK	hard disk
DL	DEC RL01 or equivalent hard disk
DX	single-density floppy disk
DY	dual-density floppy disk
CT	cassette tape
PP	paper-tape punch
PR	paper-tape reader
LP	line printer
MT	9-track magtape
VM	virtual memory
CLK	line frequency clock
TD	R7912 Transient Digitizer
DPO	Digitizing Oscilloscope

System Device. The system device is that storage peripheral (the device name and drive number) from which TEK SPS BASIC is loaded. The system storage device is the default device (the one used if no device is specified) for most peripheral-oriented commands. It is also the device from which nonresident commands are auto-loaded.

Instrument Logical Unit Number. Each instrument is referenced by an instrument logical unit number (ILUN). The ILUN is associated with a particular instrument by the ATTACH statement. The maximum number of instruments that can be attached at any one time is defined at system load time (when the system software is initialized). ILUNs are specified in instrument commands (GET, PUT, ATTACH, DETACH, WHEN, and IGNORE).

ILUNs may be constants, variables, or expressions. In the command syntax, they are always preceded by a pound sign (e.g., #K+3). ILUNs eliminate the need to specify the instrument type and number every time the instrument is accessed. This speeds program execution and reduces storage needs. Also, by specifying a variable as the ILUN, subroutines can be more general purpose. That is, one routine can be used with any instrument of a specific type.

Peripheral Logical Unit Number. When peripherals are used for the input or output (I/O) of program data, they are referenced by a peripheral logical unit number (PLUN). The PLUN is associated with the peripheral device or a file on the device by an OPEN statement. The maximum number of PLUNs that can be in use at any one time is defined at system load time (when the system software is initialized). The PLUN of zero is reserved by BASIC and is permanently associated with the system keyboard terminal (KB). PLUNs are specified in the program data I/O commands (READ, WRITE, INPUT, PRINT, READU, WRITEU, and some forms of GET).

PLUNs may be constants, variables, or expressions, and in the command syntax, they are always preceded by a pound sign (e.g., #PL+3). PLUNs eliminate the need to specify the file name or device name and number whenever the file or device is accessed. This speeds program execution and reduces storage needs. Also, by specifying a variable as the PLUN, subroutines can be more general purpose. That is, a single statement can output data to (or input data from) any peripheral.

File-Structured Device. A file-structured device is a mass storage peripheral device on which data is referenced (stored and retrieved) by file name.

Directory. A directory is a table of all the names of the files stored on a device, including pointers to where these files are stored.

Directory-Structured Device. A directory-structured device is a file-structured peripheral that has a directory of the files stored on it. The files are accessed by searching the directory for the file name and using the associated pointer to find the actual location of the file. These devices include:

floppy (flexible) disk
hard (cartridge) disk
virtual memory

Serial-Access Device. A serial-access device is a file-structured peripheral on which the files are stored sequentially with the file name stored in the beginning of the file rather than in a directory. The files are accessed by searching the tape linearly (either forwards or backwards), looking for the file name. These devices include:

magtape
cassette tape

Non-File-Structured Device. A non-file-structured device is a peripheral on which data cannot be referenced (stored or retrieved) by a file name. These devices include:

keyboard terminal
line printer
paper-tape punch
paper-tape reader

Files. Any collection of information stored on a peripheral device is a file. Files stored on file-structured devices must be named. Information written to a file remains in the file until the file is canceled or the device is reinitialized. The files created by BASIC are either program files or data files.

File Names. File names must be string constants or string variables. A file name consists of up to six-upper case letters or digits. The name may be followed by an optional file-name extension. The extension consists of up to three letters or digits, separated from the file name by a period. You may use a name or extension that is longer than the six or three character limit, but additional characters are ignored by the interpreter. BASIC expects files containing nonresident commands and drivers to have the extension SPS. No other types of files should have the extension SPS. BASIC provides an extension of BAS for program files if no other extension is specified. Some examples of legal file names are:

```
"PROG.BAS"
"DATA.DAT"
"A5566.ABC"
"GIN.SPS"
"TEST.Ø4"
"15565.A"
```

Program File. A program file contains a BASIC program. A regular program file is created by the SAVE command and updated by REPLACE. The contents of a program file (the BASIC program) are brought into controller memory by an OLD, OVERLAY, or CHAIN statement. A fast overlay file, which contains a BASIC program stored in a pretranslated form, is created by the OVLSAV command and is brought back into memory with an OVLOAD statement.

Data Files. A data file contains program data. It is classified by the type of data it holds (binary or ASCII), by how the data is stored (formatted or unformatted), and by how the data is accessed (sequentially or randomly).

Data can be stored as either a binary number or an ASCII character string. Numeric values are usually stored in two-word, floating-point binary format; but one command (WRITEU) can optionally store numeric values in 16-bit (one word) binary integer format. BASIC stores strings in ASCII format with eight bits (one byte) used for each character. Of course, if numeric characters are part of a string, they too are stored in ASCII format.

Sometimes other information describing the data (data descriptors) or delimiters between data elements are written into the file along with the data. When this is the case, the data file is said to be **formatted**; otherwise, the file is said to be **unformatted.** The data descriptor or delimiters require space in the file in addition to the data they accompany.

The WRITE command stores and the READ command reads both floating-point binary and ASCII data that is formatted by data descriptors. The PRINT command outputs and the INPUT command retrieves only ASCII data which is formatted by delimiters following the data items. The WRITEU command stores and the READU command reads unformatted, binary (floating-point or integer) and ASCII data.

Sequential Access. Sequential access is a method of file access in which the data is stored serially from the beginning of the file and is retrieved in the order in which it was stored.

Random Access. Random access is a method of file access in which the data can be stored or retrieved, in any order, as logical units of data called records. Each data record may consist of one or more data items, but all the data records in a file must be the same length. Records are stored and retrieved by record number. Where a record is written does not depend on where a previous record was written in the file. Similarly, where a record is read from does not depend on the position of the previously read record. In TEK SPS BASIC a random access file is called a record I/O (input/output) file.

SECTION 2

EXPRESSION EVALUATION

Numeric Expressions

All numeric expressions and arithmetic functions performed in TEK SPS BASIC software are evaluated in floating-point format. This is true even for integer arrays and waveforms. All elements in an integer array are temporarily converted to floating-point numbers before any operation involving them is performed. Constants appearing in a program are treated as floating-point values regardless of whether they have a fractional portion or not.

Notation Formats

Values are expressed in one of two ways. A value between .Ø1 and 999999, inclusive, is printed in its entirety. For other values, the number is printed in 'E' notation. The E notation may also be used when you input numbers to a program.

Some examples of numbers in E format are:

1.23E+6	which equals 123ØØØØ
25.5E-4	which equals .ØØ255
-14.Ø2E1Ø	which equals -14Ø2ØØØØØØØØ

The value following the E is a power of ten which is multiplied by the given number to produce the result. The plus sign is optional and may be omitted from the exponent.

Numeric Constants and Variables

Constants and variables are the components which all numeric expressions are built upon. A constant is simply a number appearing in an expression. In the following statement, 4.56 is a constant and also the source expression of the assignment statement:

```
5ØØ LET X = 4.56
```

Variables are labels that represent quantities. In the above statement, X is a variable, and after execution of the statement, X is set equal to the quantity 4.56. Before this statement is executed, X may have had any value, or have been previously undefined (a value of zero is assigned to a variable by BASIC until you give it a different value).

Arithmetic Expression Evaluation

Arithmetic expressions, as found in the LET, PRINT, and other commands where an expression is valid, are evaluated according to a standard precedence of operations. To illustrate this, consider the following expression:

```
LET J = LOG(X+Y)+V
```

Given that the values of X, Y, and V have been previously defined, evaluation of this expression follows in a natural order. That is, V cannot be added to the LOG(X+Y) until LOG(X+Y) is evaluated. But, LOG(X+Y) cannot be evaluated until the result of adding X to Y is found. No matter how complicated an expression may be, it can always be broken down into a series of operations involving one of the five basic math operators (addition, subtraction, multiplication, division, and exponentiation) and functions.

To avoid ambiguity in expression evaluation, TEK SPS BASIC follows a standard operator precedence. This order is as follows:

1. Expressions within parentheses are evaluated first, using the operator precedence defined below.

2. Function calls (such as LOG, EXP, etc.) are evaluated second.

3. Arithmetic operations are performed last and occur in the order defined in Table 2-1.

When several operators of the same precedence occur in sequence (such as in A+B+C-D), the expression is evaluated from **left to right**.

TABLE 2-1

Arithmetic Operator Precedence

Precedence	**Operation**	**BASIC Symbol**
1	exponentiation of a positive value	^
2	multiplication and division	* and /
3	addition and subtraction or negation	+ and -

Notice that in exponentiation, the value being raised to a power (the root) must be positive. Even though the exponent can be negative, the root must be greater than zero. If it is not, a warning error is issued and the absolute value of the root is used. BASIC limits exponentiation to positive values in order to allow the exponent to be fractional.

The arrangement of operators, subexpressions (expressions within expressions), and parentheses follows these rules:

1. Two subexpressions must be separated by an operator (^,*,/,+, or -).

2. Two operators may not be adjacent.

3. Three of the five operators (^,*, and /) must always be preceded and followed by a subexpression. The other two operators (+ and -) may optionally be preceded by a subexpression but must always be followed by a subexpression.

4. Parentheses must be used in pairs; for each left parenthesis there must be a matching right parenthesis, and vice versa.

The following shows some examples of illegal expressions and why they are illegal:

Illegal	**Why illegal**	**Correct**
(A+B)(C-D)	missing operator	(A+B)*(C-D)
A/-B	two adjacent operators	A/(-B)
A/(*B)	missing expression	A/(N*B)
A/(B+C)*D)	unmatched right parenthesis	A/((B+C)*D)

When a minus sign is not preceded by a subexpression, BASIC interprets the minus sign to mean "change the sign of the subexpression that follows." In each of these expressions:

```
-A      -A+B      B*(-A)
```

the minus sign in front of the A changes the sign of the subexpression A. The minus sign in the expression C/(-A^B) changes the sign of the result of the subexpression A^B, not just A.

You may also omit the subexpression preceding a plus sign, but, if so, BASIC takes no action with the plus sign. The expression that follows the plus sign is unaffected by its presence. Though not illegal, such a plus sign can just as well be omitted.

Let's consider a complex expression and discuss how it is evaluated in BASIC. The statement is:

```
LET I = LOG(X+Y) + V / F^G + H
```

First, the value of X+Y is computed, since this subexpression is in parentheses. The LOG of this sum is then computed. The next highest operation is F^G. This evaluation is followed by the division of the variable V by the result of F^G. The entire expression has now been reduced to LET I = M + N + H where M is the value of LOG(X+Y), N is the result of V/F^G, and H is the original value of variable H. Evaluation now proceeds from left to right, summing the three remaining values to produce the result I.

In some cases, the normal order of operator precedence does not permit the desired solution for the problem under study. When this occurs, normal operator precedence can be manipulated by the use of parentheses, since operations enclosed in parentheses are always evaluated first.

When parentheses are nested (one or more pairs of parentheses enclosed in an outer pair), the expression within the innermost pair of parentheses is evaluated before the outer pairs. To help understand this, consider the following expression:

```
LET I =(LOG(X+Y) + V) / (F^(G + H))
```

In this example, evaluation begins with the leftmost, innermost parenthetically enclosed expression, X + Y. The sum of this expression is then operated on by the LOG function, and to this result the value of V is added.

Now, the innermost parenthetical expression not yet evaluated is G+H. This sum is computed and the value of F is raised to this power. This result (F^(G+H)) is used to divide the result of the expression (LOG(X+Y)+V), and the final result is placed in variable I.

Here's another way of looking at parentheses, and how they can alter the normal order of evaluation. Consider the following examples:

```
1.        X = A+B*C
2.        Y = J*K/L*M
3.        Z = A/B/C
```

In example one, the product of B*C is added to the value of A. If you wish to add A and B first, then multiply by C, the statement to use is X = (A+B)*C.

In example two, the product of J*K is divided by L, and this quotient is then multiplied by M. If you wanted to divide the product J*K by the result of L*M, the statement should be Y=(J*K)/L*M).

In example three, the quotient of A/B is divided by C. If you want to divide A by the quotient of B/C, use the statement Z=A/(B/C).

Array Expression Evaluation

TEK SPS BASIC allows you to combine simple variables with arrays and waveforms in expressions. In the following discussion, waveforms may be substituted wherever arrays appear.

There are three basic forms of array assignments. These are:

Simple numeric variable = array expression. When a simple numeric variable (a variable that has not been dimensioned) is the destination of a waveform or array expression, the variable is auto-dimensioned to an array the size of the source array and set equal to the element-by-element result of the expression.

Array = expression. Every element in the destination array is set equal to the value of the expression.

Array = array expression. This is similar to the first case above. The destination array and all arrays in the source expression must be of the same size.

For a two-dimensional array, with the first subscript considered to be the row number of a matrix and the second subscript considered to be the column number, the array is filled row by row. As an example, consider this routine that fills an 8-element one-dimensional array, A, with the numbers 0 through 7. (Each element is set equal to its subscript number.) Then a two-dimensional array B is set equal to A.

```
10 DIM A(7),B(1,3)
20 FOR K=0 TO 7
30 A(K)=K
40 NEXT K
50 B=A
60 PRINT "FIRST ROW:"
70 PRINT B(0,0:3)
80 PRINT "SECOND ROW:"
90 PRINT B(1,0:3)
```

From the output below, you can see that the elements of B were assigned the values of A, element by element, row by row.

```
FIRST ROW:
              0         1         2         3
SECOND ROW:
              4         5         6         7
```

Array expressions are evaluated in the same manner as arithmetic expressions except the result is an array of numbers -- not a single number.

All the arrays (or array zones) in the expression must contain the same number of elements because the resultant array's values are calculated in a linear manner, one element at a time. The first element of the result is equal to the evaluated combination of the first elements of all the arrays in the expression and any constants or simple variables in the expression, and so on for each element. The following program demonstrates the operation.

```
100 DIM X(2),Y(2),Z(2)
110 X(Ø)=1
120 X(1)=2
130 X(2)=3
140 Y=2
150 Z=X+2*Y
```

After execution of this program, the arrays X, Y, and Z will have the following values:

```
X(Ø): 1       Y(Ø): 2       Z(Ø): 5
X(1): 2       Y(1): 2       Z(1): 6
X(2): 3       Y(2): 2       Z(2): 7
```

Line 1ØØ of the program defines the four variables (X, Y, and Z) as one-dimensional floating-point arrays of three elements each.

Lines 11Ø to 13Ø set each element in array X equal to the constant on the right of the equal sign. Line 14Ø sets all three elements of array Y equal to two. Since the array is specified as the destination and no subscript is specified, all elements in the array are operated on.

The expression in line 15Ø sets all elements in the Z array. This one statement is the equivalent of the following three statements:

```
Z(Ø)=X(Ø)+2*Y(Ø)
Z(1)=X(1)+2*Y(1)
Z(2)=X(2)+2*Y(2)
```

Had array Z not been dimensioned in line 1ØØ, auto-dimensioning would have occurred at line 15Ø, producing the same results.

Array Zones

An array zone is treated exactly like a regular array but only those elements in the zoned portion are operated on. A colon is used to delimit the boundaries of the zone. For example, the following statement creates a ten-element array (P) whose elements are equal to the first ten elements of array M:

```
P = M(0:9)
```

If variable P had been previously defined as an array, it must have been dimensioned to ten elements (for example, DIM P(9)). Array M may be dimensioned to any size equal to or greater than ten.

There are three legal variations of zoned arrays. They are:

1. array (n:m,e)
2. array (e,n:m)
3. array (n:m)

In these examples, e, n, and m are all arithmetic expressions resulting in a single value. The starting element of a zone is represented by n, and m represents the end of the zone; e is a regular subscript expression. Note that when two-dimensional arrays are referenced, only one dimension may be zoned. Hence, array (n:m,y:z) is not a valid zone.

All the rules of array expression evaluation apply when working with zoned arrays.

Although waveforms may not be zoned, the array part of the waveform may be zoned. Consider the following statements:

```
350 WAVEFORM B IS AB(511),DI,HB$,VB$
360 X=MAX(AB(25:500))
```

Statement 360, which sets variable X equal to the maximum value in the zoned portion of array AB is legal, since only the array component of the waveform is specified. Had this statement been entered as

```
360 X=MAX(B(25:500))
```

an error would have resulted because now the waveform name is specified.

Waveform Expression Evaluation

Expressions containing waveforms are treated like array expressions, except that each waveform's units and data sampling interval (DSI) are taken into account. A waveform may have a floating-point or integer array, and may be mixed with floating-point or integer arrays, waveforms, and variables in the expression.

The automatic units processing provided by TEK SPS BASIC when you use waveforms can save you time and programming effort. Before using waveform expressions, you should understand how TEK SPS BASIC processes units and data sampling intervals.

Table 2-2 shows all the combinations of expression components using waveforms, with a waveform as the destination. In the table, W1 and W2 are waveforms, A is an array equal in length to the waveform(s), and V is either a numeric variable (floating-point or integer) or a numeric constant. Note that when waveforms are mixed with arrays in waveform expressions, the result may have no units or invalid units. Invalid units are marked by a delta (Δ).

TABLE 2-2

Arithmetic Operations With Waveforms

OPERATION	RESULT
W1+V or V+W1 W1-V or V-W1 W1*V or V*W1 W1/V W1^V or V^W1 W1+A or A+W1 W1-A or A-W1 W1*A or A*W1 W1/A W1^A or A^W1 W1^W2	Result is waveform with: (1) Horizontal units same as W1's. (2) Vertical units same as W1's. (3) Data sampling interval same as W1's.

V/W1 A/W1	Result is waveform with: (1) Horizontal units same as W1's. (2) Vertical units inverse of W1's. (3) Data sampling interval same as W1's.
W1+W2 W1-W2	Result is waveform with: (1) Horizontal units same as W1's (preceded by delta if W1's horizontal units and data sampling interval are not identical to W2's). (2) Vertical units same as W1's (preceded by delta if W1's vertical units are not same as W2's.) (3) Data sampling interval same as W1's.
W1*W2	Result is waveform with: (1) Horizontal units same as W1's (preceded by delta if W1's horizontal units and data sampling interval are not identical to W2's). (2) Vertical units of W1 concatenated with those of W2. (3) Data sampling interval same as W1's.
W1/W2	Result is waveform with: (1) Horizontal units same as W1's (preceded by delta if W1's horizontal units and data sampling interval are not identical to W2's). (2) Vertical units being W1's vertical units concatenated with the inverse of W2's vertical units. (3) Data sampling interval same as W1's.

NOTE

An arithmetic function returns either a single number or an array -- never a waveform. When a waveform is an argument to an arithmetic function, the function operates on its array part only; units and data sampling interval are not associated with the result. The waveform itself is, of course, not altered **unless it is the destination of the result of the function operation.**

Since an arithmetic function never return a waveform, if W2 is a waveform, the statement

```
W2 = SIN(W2)
```

will set W2's vertical and horizontal units to null and its data sampling interval to zero. Similarly, if W3 is a waveform

```
W3 = SIN(W2)
```

will nullify W3's units and data sampling interval (DSI), even if they had been defined prior to the assignment statement. But, this second statement will not change W2's units or DSI unless they are the same variables associated with W3.

Here is an example of the automatic processing of waveform units and data sampling interval in an array expression. Suppose A1, A2 and A3 are arrays of equal size. A1 and A2 are filled with data; A3 is to hold the results. The following program associates waveforms with these arrays, assigns values to the units variables, and performs simple waveform operations. D1, D2 and D3 are the data sampling intervals. H1$, H2$, and H3$ are the string variables that hold the horizontal units. V1$, V2$, and V3$ hold the vertical units.

```
1Ø  WAVEFORM W1 IS A1,D1,H1$,V1$
2Ø  WAVEFORM W2 IS A2,D2,H2$,V2$
3Ø  WAVEFORM W3 IS A3,D3,H3$,V3$
4Ø  D1=1E-6/51.2
5Ø  D2=D1
```

```
60  H1$="S"
70  H2$=H1$
80  V1$="V"
90  V2$=V1$
100 W3=W1*W2
110 W1=W3/W2
```

Line 100 yields the waveform, W3, with units defined as follows:

```
D3 = D1 = 1E-6/51.2
H3$ = H1$ = "S"
V3$ = V1$&V2$ = "VV"
```

When line 110 is performed, the data sampling interval size and horizontal units of W1 are unchanged. However, the vertical units become:

```
V1$ = V3$&"/V" = "VV/V" = "V"
```

That is, TEK SPS BASIC software "cancels" the units.

Binary Number Limitations

Numbers in TEK SPS BASIC are represented with about 7.2 decimal digits (24 binary bits) of accuracy. Many numbers cannot be fully represented by the internal storage format and must be rounded up or truncated to the nearest 24-bit binary number.

After execution of the following BASIC statement, the value of the variable A is not actually 0.1.

```
10 LET A=.1
```

Because the number must be rounded to 24 bits, the variable A takes on a value closer to 0.10000002. For most arithmetic operations, this deviation in the 8th place is perfectly acceptable. But subtle problems can develop when these numbers are used to control iterative operations.

In Example 1, you might expect the FOR/NEXT loop to terminate after eleven iterations. It's a reasonable expectation, but an incorrect one.

Example 1

```
10 FOR A=1 TO 2 STEP .1
     .
     .
     .
50 NEXT A
```

When the variable A has an accumulated value of about 1.90000018 (on the tenth iteration) and the step value of .10000002 is added, A's new value exceeds the loop limit of 2 and the loop terminates.

When the control elements of a FOR/NEXT loop, or a loop controlled by an IF statement, can be represented exactly within 24 binary bits, there is no problem with early loop termination. Numbers that fit exactly include all integers of 7 digits or less and any negative power of 2 such as 2^{-1}=1/2, 2^{-2}=1/4, 2^{-3}=1/8, ... Most often, however, the loop control elements do not meet these criteria or are calculated in the program and are thus unknown to the programmer.

If the step value in Example 1 is used only to control the number of iterations of the FOR/NEXT loop, the following method gives good results.

Example 2

```
10 FOR A=0 TO ITP((2-1)/.1)
     .
     .
     .
50 NEXT A
```

The general form is:

```
FOR loop counter = 0 TO ITP((loop limit - loop start) / step value)
.
.
.
NEXT loop counter
```

If the loop counter is to be used as data as well as to control the loop, an additional line provides the required value and avoids potential inaccuracies that accrue because of rounding or truncation. The 3 lines of Example 3 should be used instead of the 2 lines of Example 1.

Example 3

```
10 FOR A=Ø TO ITP((2-1)/.1)
2Ø AA=1+A*.1
    .
    .
    .
6Ø NEXT A
```

The general form is then:

```
FOR loop counter = Ø TO ITP((loop limit - loop start) / step value)
LET data value = loop start + (loop counter * step value)
 .
 .
 .
NEXT loop counter
```

String Expressions

String Constants and String Variables

The main components of a string expression are string constants and string variables. A string constant appears in an expression as a sequence of ASCII characters enclosed in single or double quote marks. In the following statement, "AN 8-BIT BYTE" is a string constant:

```
1ØØ LET X$="AN 8-BIT BYTE"
```

String variables are labels that represent strings. They are distinguishe from numeric variables by the dollar sign ($). In the above statement, X$ is a string variable. After execution of the statement, X$ is set equal to the 13-character string "AN 8-BIT BYTE". Before the statement executes, X$ may be equal to any string of any length, or it may be undefined (not explicitly assigned a value by a BASIC statement). If an undefined string variable is used in an expression, BASIC will give it a default value of the null or empty string, "".

Subscripted String Variables

A one- or two-dimensional string array may be dimensioned in the same manner as a numeric array. The dimensions refer strictly to the number of strings in each dimension. There is no assumption or restriction on the length of any of the elements in the string array. Each element can be defined as a different number of ASCII characters. A single element of a string array is referred to as a subscripted string variable and may be accessed only as such. **Only the individual subscripted string variables of an array can appear in an expression; string array operations are not allowed.**

Concatenation

The ampersand (**&**) is the only string operator. It specifies the concatenation of strings. It links strings together without intervening characters. For example, if A$ equals "THE", the statement

```
B$=A$&"SIS"
```

would assign the string "THESIS" to B$.

String Expression Evaluation

The result of a string expression is a single string of ASCII characters. String expressions basically follow the same order of evaluation as numeric expressions. This order is:

1. String expressions in parentheses are evaluated first.

2. String function calls (such as CHR, SEG, and STR) are evaluated next.

3. The concatenation operation is done last, from left to right.

For example, consider the string expression in this statement:

```
LET X$ = TRM(J$&K$)&SEG(A$,1,2Ø)
```

The subexpression J$&K$ is evaluated first, forming a new string from the concatenation of J$ and K$. This new string is then trimmed of trailing blanks by the TRM function. Next, another new string is formed from the first 2Ø characters in A$ by the SEG function. Finally, the two intermediate strings are concatenated to form the string that defines X$.

SECTION 3

GETTING STARTED

Loading TEK SPS BASIC

In order to load TEK SPS BASIC from a peripheral device, the proper SPS load module of the operating system (the .LDA file) for that device must be on the medium. (Not all peripherals are supported by an SPS load module. Check the individual discussions on each device driver in the Peripheral Drivers manual to see if an SPS load module exists and what its name is.) Also, before the software can be booted, an absolute loader must have been installed on the medium by either the HOOK or HOOKQ command. (The absolute loader is a stand-alone program which, in this case, loads the operating system. The absolute loader is brought into memory by the bootstrap program.) If you follow the archiving procedure in Appendix C to make working copies of the TEK SPS BASIC System software, the correct absolute loader is installed. See the discussions on the HOOK and HOOKQ commands in Section 4 for more information.

The device name and drive number from which BASIC is loaded becomes the **system device**. This is the device and drive from which commands are autoloaded. It is also the default device and drive for many of the peripheral commands (e.g., BOOT, CANCEL, COPY, DIR, OLD, OPEN, OVERLAY, OVLOAD, READ, SAVE, WRITE, etc.) The system device driver is loaded with the operating system; it is included in the SPS .LDA file for that device.

Booting the System

After the hardward system is properly connected and powered-up, insert the medium with your copy of TEK SPS BASIC into the device for that medium. If the device has more than one drive, use the drive you perfer.

Next, follow the bootstrap procedure for the ROM bootstrap card in your controller. Some common hardware bootstrap procedures are briefly discussed in Appendix F.

When the ROM bootstrap program of the controller issues its prompt, enter the device name and the drive number from which BASIC will be loaded. For example, if your software is on a DK hard disk in drive 1, enter:

```
DK1
```

Now, depending on which absolute loader has been installed on the medium for your copy of BASIC, one of three things will happen.

1. If the SPS absolute loader was installed by the HOOK command, TEK SPS BASIC is loaded automatically. This is the most common situation.

2. If the DEC RT-11 absolute loader was installed by the HOOK command, the DEC RT-11 Operating System is loaded. To load TEK SPS BASIC, enter:

```
RUN LOADER
```

in response to the RT-11 prompt, a dot (.). When LOADER prints its prompt, an asterisk (*), enter the name of the SPS .LDA file for your device. (Do not type the .LDA extension.) For example, if the device is a DK hard disk, enter:

```
SPSDK
```

NOTE

> To return to the DEC RT-11 Operating System from TEK SPS BASIC, enter:
>
> ```
> BOOT
> ```
>
> This reboots the device with DEC RT-11 as the operating system.

3. If the LDA absolute loader has been installed by the HOOKQ command, any file with the .LDA extension can be loaded. To load TEK SPS BASIC, enter the name of the SPS .LDA file for your device in response to the prompting asterisk (*). (Do not enter the .LDA extension.) For example, if the device is a DK hard disk, enter:

```
SPSDK
```

Initializing the Software

Once TEK SPS BASIC has been loaded into the controller memory, an initialization routine begins. This routine executes only once, and then deletes itself, turning control over to Resident BASIC. The initialization process uses a set of parameters to define the size and capabilities of Resident BASIC.

The initialization routine looks for a file named "SYSBLD.DEF" on the system device. If the file is there, the user-defined parameters in the file are used to initialize the system. You can create this file by executing the SYSBLD command and answering the questions it asks. This allows you to optimize your system's use of controller memory space to fit your needs. See the SYSBLD command description in Section 4 for complete documentation.

If the file is not there, an internal list of default parameters is used to initialize a system which has or allows:

* IEEE 488 capabilities
* string functions
* graphic capabilities
* no patch area
* 6 peripheral logical unit numbers
* 8 instrument logical unit numbers
* 4 peripheral drivers
* 4 instrument drivers
* 6 nonresident commands
* clock frequency of 6Ø Hz
* graphics mode keyboard driver

Extended memory (XM) systems have an additional default parameter which allocates all of the extended memory for array storage and none of the extended memory for use as a peripheral with the VM Virtual Memory driver.

When the initialization process is complete, the version and release numbers of the BASIC Monitor and the number of words of free memory is printed on the system terminal. Then, to tell you that BASIC is in idle mode and ready for your instructions,

```
READY
*
```

is printed on the terminal.

NOTE

> Before continuing, it is strongly urged that you make a complete copy of your TEK SPS software if you have not already done so. The original copy should be placed in a safe location, and used only to produce additional copies in the event that your working copy is damaged. Appendix B contains information about how to copy TEK SPS BASIC.

How to Enter a Program

BASIC programs are usually entered from the terminal. Programs are a series of lines of text, each with its own unique line number. Each line of text contains one or more commands, instructing the system to take some action. The syntax of all the system commands appears in Section 4 of this manual.

BASIC is primarily a free-form language. That is, there are no specified columns or positions in a line set aside for special purposes. Generally, when a line is entered, the inclusion or omission of spaces is unimportant except that at least one space must follow the command name and any keywords in the command (such as THEN, AS, etc.).

If you make a mistake while typing in a line, you can use the RUBOUT or DELETE key on the terminal to delete the incorrect character. Each time the RUBOUT key is pressed, one character is deleted from the line.

If you decide it's easier to delete the whole line, type in Control-U. (Press the CONTROL key and U at the same time.) This deletes the entire line; everything up to but not including the last carriage return is deleted.

The maximum number of characters you can enter on a single line (before typing a carriage return) is 79. If you type more than 79 characters, the extra ones are not echoed (displayed on the terminal). BASIC waits for you to enter a carriage return, a Control-U, rubout, or Control-P. Eighty characters, including the terminating carriage return, is the most you can put on a single line.

A carriage return must terminate every line entered.

Running a Program

Let's try entering and running a simple program. The sample program opens a file on the system device and writes out the square roots of values entered from the terminal. Here's the program:

```
100 OPEN #1 AS "EXAMPL.TST" FOR WRITE
110 PRINT "ENTER A VALUE, NEGATIVE NUMBER TERMINATES"
120 INPUT X
130 IF X<0 THEN 160
140 WRITE #1,X,SQR(X)
150 GOTO 120
160 CLOSE #1
170 OPEN #1 AS "EXAMPL.TST" FOR READ
180 EOF #1 GOTO 220
190 READ #1,X,SQ
200 PRINT "THE SQUARE ROOT OF";X;" IS";SQ
210 GOTO 190
220 CLOSE #1
230 STOP
```

This program, while quite simple, demonstrates some of the fundamental basics of programming: input, computation, and output. Let's go over the program, line by line, and see what it does.

The first statement prepares a file on the system device to receive output. The file name is "EXAMPL.TST", and its peripheral logical unit number (PLUN) is 1. The file name is necessary when storing data on all peripherals except devices like a line printer or the paper-tape reader/punch. The name distinguishes this file from any other files on the peripheral. The FOR WRITE part of the statement tells the software that you will be sending information to the file, and not reading from it.

Line 11Ø prints a message on the terminal. Since no PLUN is specified in the statement, the terminal is assumed to be the destination of the message. This statement could have been written as

```
110 PRINT #0, "ENTER...
```

or

```
110 PRINT #K, "ENTER...
```

where variable K is equal to Ø. In any case, if the PLUN is not specified or is zero, the output goes to the terminal. The message informs the operator (you) to enter a number from the keyboard. If a negative number is entered (a number preceded by a minus sign), it signals the program to stop accepting numbers from the terminal.

Line 12Ø is an INPUT statement. When the INPUT statement is executed, a question mark (?) is printed at the terminal, prompting you to go ahead and enter the number. Here, the variable X is assigned whatever value you entered from the terminal. Like the PRINT statement in line 11Ø, no PLUN is specified in the INPUT statement. The default device for the INPUT statement is always the keyboard.

The test to determine whether to continue or end the program is made in the IF statement at line 13Ø. Here, the variable X is compared with Ø. The characters "X<Ø" mean "X is less than Ø". If so, program control transfers to line number 16Ø. If the test is false (X is equal to or greater than Ø), control goes to the next line in sequence, 14Ø in this case.

The WRITE statement in line 14Ø lists two expressions: X and SQR(X). The PLUN is specified as #1, telling BASIC that the output of the WRITE statement should go to PLUN #1, "EXAMPL.TST". No output appears on the terminal this time. Instead, the floating-point binary representations of the number you input for X and the results of the square root function with X as the argument, are output to the file.

Line 15Ø, a GOTO statement, tells BASIC to transfer control back to the INPUT statement at line 12Ø, where the sequence starts all over again. If you enter a negative number at the INPUT statement, control goes to line 16Ø. Here, the CLOSE statement ends the file "EXAMPL.TST".

In order to read the contents of the file, line 17Ø opens the file again, only this time the keywords FOR READ are used.

We won't know how much data will be output to the file, so we use an EOF (End of File) statement in line 18Ø to tell BASIC when to stop reading from the file. The line number (22Ø) following the keyword GOTO in the EOF statement tells BASIC to transfer to line 22Ø when all the data has been read from PLUN #1.

Line 19Ø reads the data. This statement reads two floating-point, binary numbers from the file (PLUN #1) and stores them in the variables X and SQ.

After data is input from a file, it can be used by a BASIC program. All we do here is print a line of output on the terminal with a PRINT statement in line 2ØØ. (Since no PLUN is specified, the output goes to the terminal.) The PRINT statement outputs the values of X and SQ (the numbers read from the file) as ASCII characters representing the decimal values for those numbers. Besides the X and SQ, line 2ØØ prints string constants that label the output. If you had INPUT a 4 in response to line 12Ø, the floating-point binary equivalent for 4 and 2 (its square root) would have been stored by line 14Ø and read into the variables X and SQ by line 19Ø. So, if a 4 is INPUT in line 12Ø, line 2ØØ will PRINT the following string characters:

```
THE SQUARE ROOT OF 4 IS 2
```

Line 21Ø, another GOTO statement, transfers program control back to line 19Ø. When the file has no more data, control passes to line 22Ø which closes the file.

The STOP statement at line 23Ø causes the message STOP AT LINE 23Ø to be printed on the terminal and returns the system to the idle mode.

Try this program yourself. To get it going, simply type RUN (followed by a carriage return, of course).

Text Manipulation in Immediate Mode

We'll assume that you did enter and RUN the practice program shown earlier and that it is still in the controller memory. Let's try altering it some to give you a few ideas about how to manipulate a program with immediate-mode commands.

For instance, to change one word of the message that is printed by the program, you could enter:

```
CHANGE 11Ø,"VALUE","NUMBER"
```

which makes line 11Ø:

```
11Ø PRINT "ENTER A NUMBER, NEGATIVE NUMBER TERMINATES"
```

Also, if you prefer to terminate a program with a RETURN instead of a STOP (to keep pending interrupts active), you could change line 23Ø. Just reenter it as:

```
23Ø RETURN
```

The new line 23Ø replaces the old 23Ø.

Now suppose you want to save this program in a program file. Typing:

```
SAVE "ROOT"
```

writes the program in a program file named "ROOT.BAS" (SAVE supplies the ".BAS" extension by default) on the system device. Of course, you still have the program in memory. If you want to look at it just type:

```
LIST
```

and the program listing is displayed on the terminal. But now you have a copy of the program stored in a file as well.

To prove it to yourself, try this. First type:

```
DELETE TEXT
```

to remove the program from memory. Now if you type:

```
          LIST
```

no program is listed. All that is displayed is the regular message:

```
          READY
          *
```

meaning that the system is in idle mode waiting for instructions. To bring a copy of the program back into memory, enter:

```
          OLD "ROOT"
```

Notice that OLD also assumes a default file name extension of ".BAS". When you type:

```
          LIST
```

the program is once again displayed.

If you try to RUN the program again, however, you will get a message that there is a P5 error in line 1ØØ. Looking up the meaning of the error and looking at line 1ØØ, you learn that you can't OPEN the file "EXAMPL.TST" for WRITE again. Before you can RUN the program another time, you'll have to CANCEL the file (remove it from the device). You can do this in immediate mode by typing:

```
          CANCEL "EXAMPL.TST"
```

but you would have to do this every time you wanted to execute the program again. A better idea is to add a line to the program, in front of line 1ØØ, that CANCELs the file. So, instead type:

```
          9Ø CANCEL "EXAMPL.TST"
```

Now you could run the program repeatedly with no P5 errors. But what of the copy of the program in the program file named "ROOT.BAS"? It doesn't have a line 9Ø. To update the copy of the program in the file, enter:

```
          REPLACE "ROOT"
```

The old file with that name on the system device is canceled and all of the program in memory is written to a new file with the same name. (REPLACE, like OLD and SAVE, assumes the default file name extension of ".BAS".)

To remove both files from the system device, type:

```
CANCEL "ROOT.BAS", "EXAMPL.TST"
```

Notice that you must include the .BAS extension for the file "ROOT.BAS". CANCEL assumes no file name extension.

Making the Most of Memory Space

The less space your program requires, the more room you have for data. Here are some ways to make a large program take up less memory space.

Use Subroutines

Make redundant lines of code into a subroutine and call the subroutine with a GOSUB statement as it is needed.

RELEASE Nonresident Commands and Drivers

Memory space can be gained by removing all the nonresident commands and drivers by executing a RELEASE ALL statement. This releases all nonresident commands and all drivers except the system device driver and the keyboard driver. If a driver can't be removed because an instrument is ATTACHed or a file is OPEN, a warning error is issued, but all modules that can be RELEASEd are removed.

Execute GETFREE

Before GETFREE calculates the amount of free memory, it compresses the string storage area. So, just executing GETFREE can make more memory available. (Executing STATUS also compresses the string storage area.)

Remove all REM statements

If you are careful never to transfer program control to a REM statement, you can decrease the size of a program by deleting all its REM statements before executing your program. See the discussion on the CHANGE command in Section 4 for an example of how to remove all the REM statements.

If you have the storage space, you may want to store two versions of a program: one with REM statements for documentation and one without, for execution. As modifications are made to the program, update the program file with the REM statements. Then, delete the REM statements from a copy of the updated program before using it to replace the contents of the program file used for execution.

Break Your Program into Segments

Make only part of your program resident at a time by using CHAIN, OVERLAY, or OVLOAD (the fast overlay command). With each method, you break your program into segments. Which method you choose depends on the application. To help you decide which command to use, the actions of the program loading commands, including OLD, are compared below.

Results of Program Loading Commands

	old text	**variables**	**Scheduler**	**Clock queue**	**interrupts**
OLD	deleted	deleted	cleared	cleared	ignored
CHAIN	deleted	remain defined	cleared	cleared	ignored
OVERLAY	unchanged except lines overlaid	remain defined	unchanged	unchanged	unchanged
OVLOAD	lines in range of fast overlay file deleted	remain defined	unchanged	unchanged	unchanged

From the table, you can see that CHAINing program segments together is generally inappropriate for applications using tasking or instrument interrupts. It is used when each section is totally independent, except for the variables. The segments are executed serially, with each segment CHAINing to (loading) the next segment to be executed. In the example below, the first program segment executes and then, just before it finishes, it CHAINs to the second segment, and so on. Here execution continues with the first line of the second segment.

```
10 REM FIRST PROGRAM SEGMENT
  .
  .
  .
500 CHAIN DX1:"PART2"
510 REM END OF FIRST SEGMENT
```

In this example, the subprogram stored in the program file named "PART2" is something like:

```
100 REM SECOND PROGRAM SEGMENT
  .
  .
  .
400 CHAIN DX1:"PART3"
410 REM END OF SECOND SEGMENT
```

Notice that the range of the line numbers in each segment can be independent of the range of the line numbers of any other segment. This is because the old program text is deleted from memory before the new text is brought in.

Use OVERLAY or OVLOAD when instrument interrupts or pending tasks must be kept active from one segment to the next. Also use OVERLAY or OVLOAD when only a portion of the program is to be replaced. Of the many ways to implement an overlay, this is one of the simplest:

```
10 REM MAIN PROGRAM
 .
 .
 .
500 REM OVERLAY A SECTION OF CODE
510 DELETE 1000,1999
```

```
520 OVERLAY DX1:"PART2"
530 GOSUB 1000
 .
 .
 .
700 REM OVERLAY ANOTHER SECTION
710 DELETE 1000,1999
720 OVERLAY DX1:"PART3"
730 GOSUB 1000
 .
 .
 .
1000 REM START OF OVERLAY AREA
 .
 .
 .
1999 REM END OF OVERLAY AREA
2000 RETURN
```

Here, a block of line numbers (1000 to 1999) is dedicated as the program's overlay area. Then, before the main program loads an overlay file, the old text in the overlay area is deleted because the lines of new text might not overlay all the lines of old text. After the overlay section is loaded, the main program calls it as a subroutine which means that a line 1000 must exist in each overlay file.

For this example, the line numbers in each of the overlay files ("PART2" and "PART3") should lie in the range of the overlay area -- between 1000 and 1999. When using the OVERLAY command you needn't always meet this condition. You may, in fact, want some lines of an overlay file to replace or intermix with lines of the existing code, but this requires careful programming. However, when you use the fast overlay command, OVLOAD, you should limit the line number range in the overlay file to the range of the overlay area of the main program. Before it loads the new program segment, OVLOAD deletes from the main program any lines of text whose line numbers are in the range of the line numbers in the overlay file.

Instrument Communication

The heart of a signal processing system is the system's ability to acquire data from instruments. Six instrument commands in TEK SPS BASIC make data acquisition a straight-forward operation: ATTACH, DETACH, GET, PUT, WHEN, and IGNORE.

Fundamental Operations

There are four fundamental operations involved in data gathering. These operations and the corresponding commands are explained below.

Attaching an instrument. Usually, all the instruments are connected to the controller via hardware data paths. The software, however, has no means of communication with an instrument until the device has been logically associated with a driver and an instrument logical unit number (ILUN). This association is made with the ATTACH statement. Once an instrument is attached, its ILUN is used when referencing that particular device.

Getting data from the instrument. The actual acquisition of data from an instrument is accomplished with the GET statement. The GET statement tells the instrument driver what information to get and where to put the data it gets. The particular instrument is referenced by its ILUN, assigned to the device with the ATTACH statement.

The actual arguments used in the GET command vary from instrument driver to instrument driver. The driver manual for the particular instrument you are working with describes in detail how to communicate with the device.

With some instrument drivers, the GET command can also be used as a direct link between an instrument and a peripheral device. Here, the destination is the peripheral logical unit number (PLUN) of a file on some peripheral device. The data goes directly from the instrument to the peripheral. This acquisition procedure is known as data-logging.

Putting data into the instrument. The BASIC command for sending data or instructions to an instrument is the PUT command. This statement directs the instrument to take some action. In the case of some digital oscilloscopes, data in the form of waveforms or messages can be sent to the device for display. The driver manual for your particular instrument describes what kinds of data can be sent to the device.

Detaching an instrument. Detaching is the opposite of attaching. When an instrument is detached, the ILUN is no longer associated with any instrument, and can be reassigned if necessary. Once an instrument is detached, no further communications can take place with it until it is attached again.

Interrupt-Driven Programs

In acquiring information from an instrument, it is sometimes necessary to wait until some event occurs in the instrument that makes meaningful information available for processing. A service request from an IEEE 488 device is one such event and the pushing of a call button on a Digitizing Oscilloscope is another. To avoid making the controller sit idle until that event occurs, BASIC permits programs to be **interrupt-driven**, that is, to let instrument events (interrupts) control the flow of the program. Efficiency of programs can be greatly increased when the controller can turn its attention from routine internal processing to instrument communication on signal from the instrument.

In BASIC, two commands regulate the communication of interrupt conditions from instruments to the controller. The WHEN command permits an instrument to interrupt and specifies what should be done when an interrupt occurs. The IGNORE command disables the interruption.

Enabling an interrupt with the WHEN command. The WHEN command lets the user tell the controller how to respond to a particular interrupt from a particular instrument and assigns to that response a priority indicating its relative importance in the processing being done. The response is determined by the user with an interrupt-handling subroutine entered as part of the current BASIC program.

Before control can pass to that subroutine, three things must happen:

1. A WHEN command must be executed which enables the program to recognise the interrupt from the instrument when the event occurs.

2. The interrupting event must occur.

3. The priority of the running program must be less than the priority assigned to the interrupt-handling subroutine by the WHEN statement.

When those three requirements are met, the current program finishes the command that was executing when the interrupt was signaled. Next, the location of the next command in that part of the program is stored. Control then passes to the interrupt-handling subroutine's first line, which is executed.

If no WHEN statement for a given interrupt has been executed, that interrupt is ignored. If the interrupt specified in a WHEN statement does not occur, the WHEN does not change the flow of program control. If the priority of the interrupt-handling subroutine is less than or equal to the priority of the currently running program, the program continues to run until it completes or until its priority is lowered to less than the priority of the interrupt-handling routine.

Once control has been given to the interrupt-handling routine, it executes either until a RETURN statement is found or until another interrupt whose routine has been assigned a higher priority occurs. In the first case, the program which was suspended to let the interrupt-handling routine execute is resumed. In the second case, the interrupt-handling routine itself is interrupted, the location of its next command to be executed is stored, and the higher-priority routine given control. Because WHEN statements can specify any of several interrupting conditions for each instrument at different priorities, BASIC is structured to permit more than a dozen levels of suspended routines to be remembered and restored in their prioritized order. Of course, if a WHEN has been executed and no other program is running when the specified condition occurs, the interrupt-handling routine immediately receives control of the computer's resources.

Disabling interrupts with the IGNORE command. An IGNORE command essentially cancels the action of a WHEN statement that specifies the same instrument and interrupting event.

The ABORT command or a fatal error in a task disables the action of all WHEN statements associated with that task number. END, STOP and Control-P cancel the action of all previously executed WHEN statements.

Data File Structures

Files are organized collections of stored information: either BASIC programs, or data. Files are stored on peripheral devices such as disks, magnetic tapes, or paper tapes. The potential length of any file is limited only by the amount of storage available on the selected peripheral.

The types of information that can be written to a data file are: 1) floating-point or integer numbers, 2) floating-point or integer arrays, 3) floating-point waveforms, 4) ASCII strings, and 5) binary data fetched directly from a specified instrument. Since these data types can be mixed in one file, it is necessary for the user to know in what order the different pieces of information are stored.

There are several ways of getting information to a data file. Three BASIC commands: WRITE, WRITEU, and PRINT can be used to transfer data from a program to a file. These commands and their output are summarized below:

Summary of Output Commands

Command	Type of Output	Format of Output
WRITE	numeric expressions, array expressions, waveform expressions, and string expressions.	Floating-point binary values and ASCII characters formatted by data descriptors.
PRINT	Same as WRITE plus string arrays. (Numeric data is converted to ASCII strings.)	ASCII characters formatted by a carriage return at the end of each line of output. (Same format as when printed to terminal, with spaces as fillers.)
WRITEU	Same as WRITE except waveform expressions are not allowed.	Unformatted floating-point and integer binary values and ASCII characters.

If data is written to a file with a WRITE statement, it must be read with a READ statement. Similarly, data output with a PRINT statement must be read with an INPUT statement, while data output with a WRITEU statement must be read with a READU statement. The INPUT command can accept data from either the terminal or a file. The READ and READU command can only read information from a file. Likewise, PRINT can output data to either the terminal or a file, while the WRITE and WRITEU commands write only to a file. The three input commands and what they read are summarized below:

Summary of Input Commands

Command	**Type of Input**	**Format of Data**
READ	Numeric variables, arrays, waveforms, and string variables.	Floating-point binary values and ASCII characters formatted by data descriptors.
INPUT	Same as for READ (Numeric strings are converted to numeric data.)	ASCII character strings followed by a carriage return. (Numeric strings may be followed by a comma.)
READU	Same as READ except waveforms are not allowed.	Unformatted floating-point and integer binary values and ASCII characters.

Sequential-Access Files

A sequential-access file is a file with data written serially, from the beginning of the file. The data must be read in the same order in which it was written. For this reason, a sequential-access file is also called a serial file. For simplicity, you might think of this type of file as being written or read, data item by data item.

A TEK SPS BASIC sequential-access file is created by an OPEN FOR WRITE statement. It is then filled with data by the WRITE, PRINT, or WRITEU command. Once it is closed, with either a CLOSE or END statement, no more information can be written to that file. It can only be reOPENed FOR READ.

Let's take a look at the contents of a data file created by the following program.

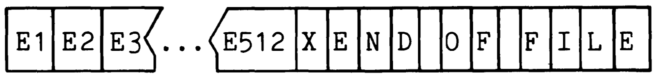
```
100 OPEN #1 AS "EXAMPL.03" FOR WRITE
110 DIM A(511)
120 X=5
```

```
12Ø X=5
13Ø A=RND(A)*X
14Ø WRITE #1,A,X,"END OF FILE"
15Ø CLOSE #1
```

This simple program first opens a file called EXAMPL.Ø3 on the system device. It then creates a 512 element array, A. Variable X is given a value of 5 in line 12Ø. This variable is used in the expression in line 13Ø to fill array A with random numbers between Ø and 5.

The WRITE statement in line 14Ø writes the array, variable X, and the message into the file. The file is closed for further writing by the CLOSE statement in line 15Ø.

After this program has executed, the contents of the file can be pictured as follows:



Each box in the picture represents one piece of data. E1 through E512 represent the 512 elements of array A. Each of these entries is a 32-bit floating-point number. X is the 32-bit value of variable X. The message is written as a series of ASCII characters, each eight bits in length.

Because the WRITE command is used, as the data is actually written onto the peripheral, each item of information (such as an array, a number, or string) is preceded by a data descriptor. These descriptors allow BASIC to know what type of information is in the file. The descriptors are not accessible to the BASIC program, however. It is up to the programmer to know in what order data is stored in the file. Appendix E contains more information about data descriptors and output formats.

If PRINT is used, all the data is output as ASCII characters. Numeric data is converted to ASCII strings that represent the decimal equivalent of the numeric data. These are called numeric strings.

PRINT does not output data descriptors but it does output a carriage return at the end of each line of output. The INPUT command, which reads data from a file filled by PRINT, expects each data item (an ASCII string) to be followed by a carriage return. (Numeric strings may optionally be followed by a comma instead of a carriage return.) But PRINT is intended mainly for display of data. It does not automatically output a carriage return after each data item. It only automatically outputs a carriage return after each line. For this reason you cannot simply PRINT data to a file if you intend to later INPUT that data back to a program. The discussion on the INPUT command in Section 4 shows how to PRINT data to a file if you intend to INPUT that data.

If WRITEU is used, data descriptors are not written out. This provides compatibility with DEC RT-11 FORTRAN programs. READU can be used to read files written using WRITEU and other files containing unformatted binary and ASCII. See the SYSBLD command in Section 4 for an example of how to read such a file using READU.

Random Access Files

A random access file is a file in which data is stored as data records that can be written or read in any order. In TEK SPS BASIC VØ2, a random access file is called a record I/O (input/output) file.

A record I/O file is created by a DEFINE statement which determines the size of the file. Then, when it is OPENed FOR UPDATE, it can be written to by the record I/O form of WRITEU or read by the record I/O form of READU. Once it is closed, by either a CLOSE or an END statement, it can be reOPENed FOR UPDATE to read, change, or even add more data providing there is room in the file for the record specified.

Since the data is accessed as a collection of data items, called a data record, and not as single data items, it is up to the user to know the type, order, and size of the data items in the record as well as the record number of the record accessed. Programs demonstrating the use of the record I/O forms of READU and WRITEU can be found in the command descriptions of those commands in Section 4.

Adding to a Sequential-Access File

Once a sequential-access file has been closed, no more information can be added to that file. However, it is possible, through programming, to create a new file consisting of the old data and any additional information required. The following program demonstrates how to add data to a numeric data file.

```
100 OPEN #1 AS O$ FOR READ
110 OPEN #2 AS "SCRTCH" FOR WRITE WITH 2
120 EOF #1 GOTO 170
130 DELETE A
140 READ #1,A
150 WRITE #2,A
160 GOTO 130
170 WRITE #2,N
180 CLOSE #1
190 CLOSE #2
200 CANCEL O$
210 RENAME "SCRTCH" TO O$
220 END
```

The original file name is stored in the string O$. Line 100 OPENs this file for READ. Another file, called "SCRTCH," is OPENed for WRITE at line 110. The EOF statement at line 120 sends program control to line 170 when the data in the original file is exhausted. The technique of using the EOF statement allows you to read a file of any length, without knowing beforehand the exact length of the file. (The EOF command does not cause a program branch until the data file is exhausted.)

The DELETE statement at line 130 assures that variable A is not an array. This is necessary because some of the data in the file might be arrays. If so, variable A, being a simple numeric variable, will be automatically dimensioned to the correct size when the array is read.

The READ statement at line 140 reads the next item of information from the original file. It is immediately rewritten into the scratch file at line 150. After the write, control goes back to the DELETE statement and the read and write process takes place again.

When all the data has been read from the original file, control passes to line 170 (because of the EOF statement in line 120), and the new information, N, is written into the new file. Both files are CLOSEd in lines 180 and 190, and the original file is deleted from the peripheral by the CANCEL statement at line 200. Line 210 renames the scratch file to the original file name contained in O$.

Note that these files used the WRITE and READ statements exclusively. Had the original file been created with one or more PRINT statements, this program would not work. Likewise, if strings had been output to the original file, the program would have to know at what point these strings occur, and specify a string variable in the READ and WRITE statements.

Reading in Arrays

An array can be read into memory from a peripheral device in many ways. The entire array, zones of the array, or individual elements can be read at one time. The action taken by BASIC is dependent on where the data pointer (the position in the file of the next piece of data) is located, and on the type of variable that will receive the data. Let's look at some examples.

First, consider a file that contains a ten-element array. The file has just been opened for read, and the data pointer is positioned at the beginning of the file. The data descriptor tells BASIC that the information is an array of ten elements. Now, the following statement is executed:

```
READ #1,T
```

After this statement is executed, the simple numeric variable T is auto-dimensioned to nine (ten elements, subscripts zero through nine), and the data pointer is positioned at the end of the file. Array T now contains all ten elements of the array in file #1.

Let's take another example using the same file. Here, two variables are going to be used in the READ statement, each a five-element array. The program follows:

```
DIM A(4),B(4)
READ #1,A,B
```

Assuming that the file has just been opened, and the data pointer is at the beginning of the file, the execution of these two statements fill both five-element arrays with data. The first five data elements in the file go to array A, and the last five elements go to array B.

Reading arrays into subscripted variables. Individual elements of an array can be read, one at a time, by using a subscripted variable in the READ statement. Using the same example file, the following statements read in two elements of the array, leaving the data pointer at the third array element in the file. C(Ø) will get the first element of the stored array and C(3) will get the second element.

```
DIM C(5)
READ #1,C(Ø),C(3)
```

As you can see, there are many ways to read in arrays from a peripheral file. Giving examples of each possible READ statement is impossible. However, there are some rules you should know.

End of file conditions. If a READ statement needs more data to assign to its arguments than is present in the file being read from, a fatal error is issued. This can happen, for instance, if an array specified in a READ is dimensioned to a size greater than the number of data elements left in the file or if there are more variables in the READ statement than elements in the file. The error can be avoided by executing an EOF statement prior to the READ. However, the programmer should be prepared to correct the situation in which some variables which may be used later in the program have not been assigned values from the file.

Reading single data values as an array. This condition, while by no means an error, can cause confusion. If a series of numeric expressions (not arrays or waveforms) is written to a file, it can be read in again as an array. For example, the following statements write out ten values to file #1:

```
OPEN #1 AS "FILE" FOR WRITE
WRITE #1,A,B,C,D,E
WRITE #1,T1,T2,T3,T4,T5
CLOSE #1
```

When this file is read again, all ten values in the file can be placed into one array with one READ statement. In the following example, a single READ statement, referencing one array, is all that is needed to read in the entire contents of the file:

```
OPEN #1 AS "FILE" FOR READ
DIM P(9)
READ #1,P
CLOSE #1
```

After execution of these statements, array P contains all ten values written to the file in the previous example. Had array P been dimensioned to fewer than ten elements, the array would have been filled, and the data pointer would be positioned at the next available entry in the file. If array P had been dimensioned to more than ten elements, a fatal error would have been issued. Notice that, since the data items were written out individually, P would **not** be autodimensioned by the READ statement.

SECTION 4

TEK SPS BASIC COMMANDS

This section of the manual contains a complete description of the system commands. It does not discuss optional software modules such as signal processing, graphics, and instrument drivers. These optional commands and drivers are described in separate manuals which accompany the optional software.

Each command description includes statement examples, syntax information, and a general discussion of what the command does. Many descriptions include suggestions for using the commands. Within the descriptions, information that a beginning user need not read is enclosed in bold-face square brackets; information that pertains only to extended memory (XM) systems is shaded.

Overview of System Commands

The descriptions of the TEK SPS BASIC VØ2 system commands appear in this section in alphabetical order to make each one easier to find. But for your assistance, all the system commands, with a brief description, are listed below by use. Some commands appear in more than one list. An asterisk indicates a nonresident command.

System Control Commands

Commands in this category allow you to edit and list program text, load and release modules, change system status and priority, bootstrap another device, and enter and run programs.

*	**ABORT**	Terminates a single task.
*	**BOOT**	Reloads BASIC system software from a peripheral device.
*	**CHAIN**	Deletes the current program and loads and starts executing the specified new program. Does not delete variables.
*	**CHANGE**	Edits program text in memory.
	DELETE	Removes program lines, waveforms, arrays, and string arrays from memory.
*	**GETLOC**	Obtains the contents of a specified memory location.
*	**LIST**	Prints all or part of the current program text on the system terminal or specified peripheral device.
	LOAD	Loads specified drivers or commands into memory.
*	**LOCKKB**	Limits system input to Control-P while a program is running.
*	**LST**	Prints all or part of the current program text on the system terminal or specified peripheral device. FOR/NEXT loops are indented and concatenated statements are displayed with one command per line.

	OLD	Loads a new program or program segment into memory, deleting all existing text and variables.
*	**OVERLAY**	Loads a new program or program segment into memory without affecting variables. Overlays lines with matching line numbers, but does not delete other program text in memory.
*	**OVLOAD**	Performs a fast overlay of a pretranslated BASIC program segment from a file created by an OVLSAV statement.
*	**PRIORITY**	Changes the priority of a running program.
*	**PUTLOC**	Assigns a specified value to a memory location.
*	**RANDOM**	Sets seed value of the random-number generator or returns seed value.
	RELEASE	Removes nonresident commands or drivers from memory.
	REM	Allows inclusion of remarks in program listing.
*	**RENUM**	Assigns new sequential line numbers to part or all of program text in memory.
	RUN	Starts program at line having lowest line number in memory.
*	**SETDATE**	Sets the system date.
*	**SETTIME**	Sets the system time.
*	**SYSBLD**	Defines the contents of file to set system parameters at initialization time.

Program Control Commands

These commands affect and direct program flow.

	END	Terminates all program execution, closes all files, disables instrument interrupts, returns to idle mode.
*	**EOF**	Designates a program line to receive program control when data from a peripheral file is exhausted.
	FOR	Specifies start of program loop and controlling parameters.
	GOSUB	Transfers program control to a subroutine or to one of a list of subroutines.
	GOTO	Transfers program control to a specified line, or to one of a list of specified lines.
	IF	Conditionally transfers control or executes another command.
*	**IGNORE**	Prohibits change of program flow by specified instrument conditions.
*	**INPREQ**	Permits unsolicited input of data from the keyboard while a program is running.
	NEXT	Terminates FOR loop.
*	**ONERR**	Allows processing of errors in a BASIC program.
*	**RESCHEDULE**	Puts either the current task or the task on Scheduler stack back on Scheduler queue.
	RETURN	Terminates the execution of a subroutine.
*	**SCHEDULE**	Queues a subroutine for execution at a specified time or after a specified time lapse.
	STOP	Terminates program execution, disables instrument interrupts, and returns to idle mode.

*	**UNSCHEDULE**	Cancels the actions of a SCHEDULE command if the specified time has not elapsed.
*	**WAIT**	Stops execution of a program until a keyboard interrupt is received or a specified amount of time has elapsed.
*	**WHEN**	Allows specified instrument conditions to change program flow.

Variable Definition Commands

These commands allocate or reclaim storage space or assign values to variables.

*	**ATAN2**	Performs double-argument arctangent.
*	**CLEAR**	Initializes all variables and arrays to zero, string variables to null strings.
*	**DATE**	Obtains system date.
	DELETE	Removes program lines, waveforms, arrays, and string arrays from memory.
	DIM	Assigns floating-point storage space for array variables or defines string arrays.
*	**GETFREE**	Obtains the amount of free memory currently available.
*	**GETLINE**	Obtains the line number of the line being executed.
*	**GETPRI**	Obtains priority of task being executed.
*	**HASH**	Converts a string (hash key) to an index number that can be used to access an indexed list for storing and retrieving data.
	INTEGER	Allocates integer-format storage for arrays.
	LET	Assigns the value of an expression to a variable, array, waveform, or string variable.

*	**MATCH**	Obtains the index of the string array element containing the search string.
*	**RANDOM**	Sets seed value of the random-number generator or returns seed value.
*	**TIME**	Obtains system time.
*	**VERSION**	Obtains the version and release numbers of a nonresident command or driver or of the BASIC monitor.
	WAVEFORM	Associates a data array with its data sampling interval and units.

Program Data Input/Output Commands

Input/Output (I/O) commands concern data transfers between a program and a peripheral device or file.

	CLOSE	Terminates I/O with a device or file.
*	**DEFINE**	Creates a Record I/O file.
*	**EOF**	Designates a program line to receive program control when data from a file is exhausted.
*	**GETBLK**	Obtains the contents of a block from a directory-structured device.
*	**INPREQ**	Permits unsolicited input of data from the keyboard while a program is running.
*	**INPUT**	Obtains ASCII values for variables from the keyboard or other peripheral device or an ASCII file, and if the variables are numeric, translates those values to binary form.
	OPEN	Allows access to an existing data file, a new data file, or a non-file-structured peripheral device.
*	**PRINT**	Outputs ASCII information to the terminal or other peripheral device or a data file.

*	**PUTBLK**	Stores a physical block of data on a directory-structured device.
	READ	Obtains formatted binary and ASCII values for variables from a peripheral device or file.
*	**READU**	Obtains unformatted binary and ASCII values for variables from peripheral device or file.
*	**RESET**	Resets a file that is OPEN for READ to the beginning of that file.
*	**REWIND**	Rewinds serial tape devices.
*	**WRITE**	Outputs data in formatted binary and ASCII form to a peripheral device or file.
*	**WRITEU**	Transfers unformatted binary and ASCII data to peripheral device or file.

Peripheral Housekeeping Commands

Housekeeping commands allow you to transfer files between peripherals, remove files, save programs, print the directory, and in general alter the files on a device.

*	**CANCEL**	Removes specified files from a peripheral storage device.
*	**COPY**	Transfers file from one peripheral device to another device or file.
*	**DIR**	Prints on terminal or specified device a list of files stored on a peripheral device.
*	**FORMAT**	Formats a CP110 cartridge disk (a Digital Equipment Corp. cartridge disk or its equivalent).
*	**HOOK**	Writes system bootstrap program on specified peripheral device.
*	**HOOKQ**	Installs an absolute loader for .LDA files on a disk.

*	**OVLSAV**	Creates a file containing a pretranslated BASIC program segment.
*	**RENAME**	Changes the name of a file on a directory-structured device.
*	**REPLACE**	Replaces specified file on a peripheral device with program text currently in memory.
*	**SAVE**	Stores program lines on a specified peripheral device.
*	**SQUISH**	Compacts files on a disk storage device.
*	**ZERO**	Initializes the specified file-structured peripheral device.

Instrument Control Commands

Instrument control commands make data transfers between a program and an acquisition instrument. They are also used to control the operation of the instruments.

	ATTACH	Allows communication with an instrument.
	DETACH	Terminates communication with an instrument.
*	**GET**	Fetches data or status information from an instrument and stores it in specified variables or in a specified peripheral file.
*	**IGNORE**	Prohibits change of program flow by specified instrument conditions.
*	**PUT**	Sends data or status information from memory to a specified instrument.
*	**WHEN**	Allows specified instrument conditions to change program flow.

Debugging Commands

These commands help you find the causes of errors in your programs.

*	**GETFREE**	Obtains the amount of free memory currently available.
*	**GETLINE**	Obtains the line number of the line being executed.
*	**GETPRIORITY**	Obtains priority of task being executed.
*	**LIST**	Prints all or part of the current program text on the system terminal or specified peripheral device.
*	**LISTVAR**	Lists on terminal or specified device the names and dimensions of all arrays, waveforms, variables, string variables, and string arrays currently defined.
*	**LST**	Prints all or part of the current program text on the system terminal or specified peripheral device. FOR/NEXT loops are indented and concatenated statements are displayed with one command per line.
*	**PRINT**	Outputs ASCII information to a specified peripheral device. May be used to output the contents of program variables or a message to the terminal.
*	**STATUS**	Prints the current status of the system on the terminal or specified peripheral device.
*	**VARTST**	Tests for set bits.

Guide to Notation

Syntax and Descriptive Forms

The term syntax refers to the rules for allowable statement structures in a programming language. The rules for a permissible BASIC statement are shown in the syntax form for each command. This form indicates the command's delimiters (punctuation), keywords, and legal arguments plus what is required and what is optional. In the syntax form, the words describing the command's arguments tell you what is acceptable but not what is the meaning or use. To help clarify the meaning of the syntax form, most command discussions also have a descriptive form, which indicates the purpose of each syntax component. For example:

Syntax Form:

[line no.] **RANDOM** floating-point variable, floating-point variable

Descriptive Form:

[line no.] **RANDOM** high-order part of seed, low-order part of seed

The syntax form and the descriptive form work together to give you complete information on how to enter the command. However, the descriptive form is only provided to make the meaning and use of the syntax form more understandable. It should not be considered as an exact description of the syntax.

Both the syntax and descriptive forms use these few simple conventions to convey the legal variations of a command.

1. **Items enclosed in square brackets are optional.** The statement is valid if these items are omitted. For example:

Syntax Form:

[line no.] **WAIT** [expression]

Descriptive Form:

[line no.] **WAIT** [number of milliseconds]

This command may be entered in any one of the following ways:

```
        WAIT
        WAIT 5ØØ
    1ØØ WAIT
    1ØØ WAIT 5ØØ
```

2. Optional entries within optional entries cannot be used by themselves. For example:

Syntax Form:

[line no.] **STATUS** [device name[constant]:][string expression]

Descriptive Form:

[line no.] **STATUS** [device name[drive no.]:][target file name]

The device name and the string expression (target file name) are independently optional. However, the constant (drive number) may not be entered unless the device name is entered. Thus, any of these are acceptable.

```
        STATUS
    9ØØ STATUS DX1:"STAT.FIL"
        STATUS "STAT.FIL"
    75Ø STATUS LP:
```

But,

```
        STATUS 1
```

is not.

3. Stacked items enclosed in braces make up a selection list from which one item must be selected. For example:

Syntax Form:

[line no.] **CLOSE** $\left\{\begin{array}{l}\text{\#expression} \\ \textbf{ALL}\end{array}\right\}$

Descriptive Form:

[line no.] **CLOSE** $\left\{\begin{array}{l}\text{\#peripheral logical unit number} \\ \textbf{ALL}\text{ peripheral logical unit numbers}\end{array}\right\}$

Either the pound sign and expression or the keyword ALL must be specified, so either of these is legal:

```
     CLOSE #1
 55Ø CLOSE ALL
```

But, this is not:

```
 55Ø CLOSE
```

4. Stacked items enclosed in square brackets make up a selection list from which none or one may be selected. For example:

Syntax Form:

```
                  ┌variable        ┐
[line no.] DATE   │array           │
                  └string variable ┘
```

Here, legal entries would include:

```
     DATE
 1ØØ DATE D
 15Ø DATE DA$
```

5. Three dots (...) indicate that the preceding item may be repeated. For example:

Syntax Form:

```
                 {line number                                  }
[line no.] GOTO  {expression OF line number[,line number]...   }
```

When you choose the second form, the quantity of repeatable items (a comma followed by a line number) is limited only by the length of an input line (79 characters plus a carriage return).

6. **Keywords and command delimiters should be entered as shown.** For clarity, the keywords and delimiters are printed in bold in the syntax and descriptive forms.

Keywords are alphabetic symbols used in a BASIC statement. Keywords must be entered as all upper-case characters. They may not be abbreviated unless they are nonresident command names -- in which case, the first six letters are all that must be entered. For example, while the resident command, INTEGER, cannot be abbreviated, the nonresident command, SCHEDULE, can be abbreviated to SCHEDU. If a command name can be abbreviated, it will appear that way in the syntax form. The full command name appears in the descriptive form.

Delimiters are characters which separate the elements in a BASIC statement. They are the command's punctuation. The following characters are valid delimiters used by TEK SPS BASIC:

Delimiter	**Symbol**
Space	blank
Comma	,
Semicolon	;
Colon	:
Apostrophe (or Single Quote)	'
Quotation Mark	"
Parentheses	()
Angle Brackets	<>
Pound Sign	#
Equal Sign	=
At Sign	@

7. **A space must precede and follow each keyword.** Exceptions are when another delimiter is required by the syntax. In such cases the space may be omitted if it would be a redundant delimiter.

Here are some examples of when a space is redundant: 1) The TAB function keyword in the PRINT command is followed by an open parenthesis so it is legal to omit the space. 2) The DEL keyword in the CHANGE command is preceded by a comma. No space is needed here. 3) The CLEAR command has no arguments and requires no space before the carriage return. Surrounding such keywords by spaces is not wrong; but, any unnecessary spaces will not appear in the program LISTing.

8. **A line number must be a positive integer between 1 and 32767, inclusive.**

Substitution Guide Lines

To help you make a proper entry for a syntax item, the following substitution guidelines are provided in Table 4-1. The terms used are defined in Section 1 and included in the Glossary.

TABLE 4-1

SYNTAX SUBSTITUTION CHART

Specification	Allowable Substitution
array	* a floating-point array * a floating-point array zone * an integer array * an integer array zone
array expression	* an array (see list above) * any legal combination of constants, variables, arrays, waveforms, arithmetic operators, functions, and parentheses that evaluates to an array or array zone
constant	* only a numeric constant
device name	* a 2 or 3 letter mnemonic that is used to reference an instrument or peripheral device
drive number	* a positive integer (base 10) that designates which unit of the device is specified
expression	* a constant * a variable (see list below) * any legal combination of constants, variables, arrays, waveforms, arithmetic operators, functions, and parentheses that evaluates to a single numeric value
floating-point array	* a floating-point array * a floating-point array zone

SYNTAX SUBSTITUTION CHART (cont.)

Specification	**Allowable Substitution**
floating-point variable	* a simple numeric variable * an element of a floating-point array * an element of a floating-point waveform
floating-point waveform	* only a waveform associated with a floating-point array (waveforms may not be zoned)
integer array	* an integer array * an integer array zone
integer variable	* an element of an integer array * an element of an integer waveform
integer waveform	* only a waveform associated with an integer array (waveforms may not be zoned)
line number	* an integer between 1 and 32767, inclusive.
simple numeric variable	* only a simple numeric variable (not an array or waveform element)
simple string variable	* only a string variable (not a string array element)
string array	* only a string array (string arrays may not be zoned)
string constant	* characters enclosed in single or double quotes

SYNTAX SUBSTITUTION CHART (cont.)

Specification	**Allowable Substitution**
string expression	* a string constant * a string variable * an element of a string array * any legal combination of string constants, string variables, string functions, parentheses, and the string operator (&) that results in a string
string variable	* a string variable * an element of a string array
variable	* a simple numeric variable * an element of a floating-point array * an element of an integer array * an element of a floating-point waveform * an element of an integer waveform
waveform	* a waveform associated with a floating-point array * a waveform associated with an integer array
waveform expression	* a waveform (see list above) * any legal combination of constants, variables, arrays, waveforms, arithmetic operators, functions, and parentheses that evaluates to a waveform

ABORT (Nonresident)

Examples:

```
    ABORT TASK 2
900 ABORT TASK N
```

Syntax Form:

[line no.] **ABORT** [**TASK** expression]

Descriptive Form:

[line no.] **ABORT** [**TASK** task number]

Purpose:

To allow a user to terminate one task without terminating other tasks.

Discussion:

The ABORT command halts execution of the given task. It cancels the action of all WHEN statements with the specified task number. It removes any SCHEDULEd routines with that task number from the clock queue. It also removes any routines with that task number from the Scheduler queue and stack. Thus, any subprogram associated solely with the stipulated task number is not executed. (The function and parts of the Scheduler are explained in Section 1.)

If the program is in the midst of an input/output process, the I/O finishes before ABORT halts the task. Also, if ABORT is entered in the immediate mode, the currently executing command finishes before ABORT executes.

Using the Command Syntax:

The optional expression following the keyword **TASK** specifies the task number of the task to ABORT. The expression, when evaluated and rounded to an integer, must be between Ø and 126, inclusive. If the keyword TASK and the expression are omitted, the currently executing task is aborted. That is, the task associated with the ABORT statement is the task aborted. If the ABORT command is entered in immediate mode and no task number is specified, only task 127 (the immediate mode task number) is ABORTed.

ATAN2 (Nonresident)

Examples:

```
150 ATAN2 A,B,C
160 ATAN2 B,X,Z(J)
170 ATAN2 A(5:15),B(20:30),C(0:10)
```

Syntax Form:

```
                     (floating-point variable)   (floating-point variable)
[line no.] ATAN2     {floating-point array    } , {floating-point array    } ,
                     (floating-point waveform)   (floating-point waveform)

                     (floating-point variable)
                     {floating-point array    }
                     (floating-point waveform)
```

Descriptive Form:

```
[line no.] ATAN2 real source data,imaginary source data,
                 target for arctangent of imaginary/real
```

Purpose:

To perform a double-argument arctangent operation.

Discussion:

This command computes the arctangent of the quotient of the second argument divided by the first argument, and stores the result in the third. For example, if the three arguments are A, B, and C respectively, the arctangent of B/A is stored in variable C. The answer is in the range of ± pi radians. (The arctangent function (ATN) has only half this range, ± pi/2 radians.)

Assuming the statement,

```
ATAN2 A,B,C
```

if the third argument (the target, C) is a waveform, its units and data sampling interval (DSI) are set as follows:

1. If A is a waveform,
   C's vertical units = "RAD"
   C's horizontal units = A's horizontal units
   C's DSI = A's DSI

2. If A is not a waveform and B is a waveform,
   C's vertical units = "RAD"
   C's horizontal units = B's horizontal units
   C's DSI = B's DSI

3. If neither A nor B are waveforms,
   C's vertical units = "RAD"
   C's horizontal units = null string
   C's DSI = 1

A warning error is generated if only one of the source arguments is a waveform (as in the second case above). A warning error is also generated if the source arguments are waveforms but their units and data sampling intervals are not identical.

A warning error is issued if both of the source variables are zero. In this case, the target is set to zero. With A and B the source variables and C the target, consider these examples. If A,B, and C are floating-point variables and both A and B equal zero, a warning error is issued and C is set to zero. If A,B, and C are arrays of the same size, for each array index, I, where both A(I) and B(I) equal zero, a separate warning error is issued and C(I) is set to zero.

Using the Syntax Options:

Arrays and waveforms may be used together as arguments. Note, only floating-point arrays or waveforms containing floating-point arrays may be specified. All arguments must be of the same length.

ATTACH

Examples:

```
100 ATTACH #N AS DPO3:
210 ATTACH #1 AS INS7,3:WITH 4,5 @0
```

Syntax Form:

```
[line no.] ATTACH #expression AS device name[constant[,constant]]:
                 [[WITH expression[,expression]...] @expression]
```

Descriptive Form:

```
                                      ┌┌hardware unit number                                ┐┐
[line no.] ATTACH #ilun AS device name│┤                                                    ├│:
                                      └└primary address[,secondary address of mainframe]  ┘┘
                  [[WITH secondary address of plug-in [,secondary address of plug-in]...]
                  @ IEEE 488 interface number]
```

Purpose:

To associate an instrument logical unit number (ILUN) with a specific instrument.

Discussion:

Unless the communication is performed at a low level through the IEEE 488 Interface driver (GPI.SPS), an instrument must be ATTACHed before you can use BASIC to communicate with it. The ATTACH command associates an instrument logical unit number (ILUN) with an instrument. After that, only the ILUN is used to reference the instrument, allowing you to write general purpose data acquisition and control routines.

For the ATTACH command to function properly, the instrument must be on-line (electrically connected to the controller) and powered up. Also, the instrument driver must be in memory and the specified ILUN must not already be in use (ATTACHed to another instrument).

An ILUN and instrument are dissociated by the DETACH command.

Using the Syntax Options:

The expression following the pound sign (#) is the ILUN. When evaluated and rounded to an integer, it must be between 1 and n, inclusive, where n is the number of ILUNs specified at system initialization (default value of eight).

The instrument name must be a legal two- or three-letter mnemonic. The optional constant represents either the hardware unit number (HUN) or the IEEE 488 primary address of the instrument being ATTACHed. A second optional constant representing the secondary address of an IEEE 488 instrument may follow the primary address. If the first constant is omitted, the HUN or primary address is assumed to be zero. Omitting the second constant implies there is no secondary address.

The expression following the at sign (@) is the number of the IEEE 488 Interface through which the IEEE 488 instrument is addressed. The expression, when evaluated and rounded to an integer, must be between Ø and 3, inclusive.

The optional keyword **WITH** followed by an expression list is used to associate a single ILUN with a configuration of instruments that share a common primary address and interface. Usually, the expression represents the secondary address of plug-ins, while the constant following the primary address is the secondary address of the mainframe. For example, a TEKTRONIX 7912AD Programmable Digitizer with two programmable plug-ins, such as the 7A16P and the 7B90P, can be assigned a single ILUN with a statement such as line 21Ø in the examples. In this case, the mainframe is addressed as #1 or #1;3 and the two plug-ins are addressed as #1;4 and #1;5. Used with the high-level IEEE 488 Instrument driver, INS.SPS, this form of ATTACH causes the internal routine which polls an IEEE 488 Interface when an SRQ is detected to include the plug-ins in the poll of the ATTACHed instruments. (The optional keyword WITH is not recognized by ATTACH VØ2-Ø1.)

Application Example:

The following shows the necessary order of first LOADing the instrument driver, then ATTACHing the instrument, before referencing it by an ILUN, which in this case is 2.

```
100  REM LOAD INSTRUMENT DRIVER
110  LOAD "DPO.SPS"
120  REM ASSOCIATE ILUN WITH INSTRUMENT
130  ATTACH #2 AS DPO2:
 .
 .
 .
200  REM REFERENCE INSTRUMENT BY ITS ILUN
210  GET WA FROM #2,A$
```

BOOT (Nonresident)

Examples:

```
BOOT
BOOT DX1:
```

Syntax Form:

[line no.] **BOOT** [device name[constant]:]

Descriptive Form:

[line no.] **BOOT** [name of bootable device[drive number]:]

Purpose:

To reload system software from the specified peripheral device.

Discussion:

BOOTing is the process of reinitializing the system by reloading the software. BOOTing also redefines the system peripheral device since the system device is the device and the drive from which BASIC is loaded.

The BOOT command reads in the absolute loader from the specified bootable device and passes control to that loader. The absolute loader then loads TEK SPS BASIC, an .LDA file, or the DEC RT-11 Monitor, depending on which absolute loader it is and how your system is configured.

NOTE

To BOOT a device you must first install the appropriate absolute loader in the bootstrap blocks. Read the discussion on the HOOK or HOOKQ commands for more information.

What happens when BOOT executes depends on which command -- HOOK or HOOKQ -- was used to install the absolute loader on the disk. If HOOK (without the FOR RT11 option) was used, TEK SPS BASIC is automatically loaded.

If the absolute loader was installed by HOOKQ, BASIC is not automatically loaded. Instead, a prompt (*) is printed on the terminal and the loader waits for you to enter the name of a file whose extension is .LDA. Entering SPSxx, where xx is the name of the bootable device (e.g., DX, DK, DL, or DY), loads BASIC. Notice that you **do not enter the .LDA extension or put the file name in quotes.**

[When TEK SPS BASIC is loaded under a DEC RT-11 Monitor, BOOTing loads the DEC RT-11 Monitor. Running LOADER.SAV and entering SPSxx, where xx is the name of the bootable device (e.g., DX, DK, DL, or DY), loads BASIC. See the System Peripherals manual for further discussion.]

Using the Syntax Options:

After the BOOT command executes, the named device and drive number become the system peripheral device -- the device and the drive number from which nonresident commands are auto-loaded and (usually) the default device when the device name is omitted. If the device name is omitted, the current system device is bootstrapped. If the drive number is omitted, zero is assumed.

The specified device must be one of the bootable devices such as DX, DK, DL, or DY. Unless the device is DK or DX, its driver must be loaded into memory before BOOT executes.

NOTE

The DL and DY drivers are not supported by TEK SPS BASIC V02-01.

CANCEL (Nonresident)

Examples:

```
15Ø  CANCEL DK:"TEST.DAT"
16Ø  CANCEL "PROG.BAS"
17Ø  CANCEL CT:/F,"NEW.BAS","*.DAT","LASER.*"
2ØØ  CANCEL DX1:A$&".BAS"
```

Syntax Form:

[line no.] **CANCEL** [device name[constant]:] $\left[/ \begin{Bmatrix} F \\ R \end{Bmatrix} [,] \right]$ string expression
[,string expression]...

Descriptive Form:

[line no.] **CANCEL** [device name[drive number]:][/forward or reverse switch[,]]
file name [,file name]...

Purpose:

To remove unwanted files from a peripheral device.

Discussion:

The CANCEL command logically removes the specified files from a file-structured device. For a directory-structured device, the CANCELed file names are deleted from the directory so space on the device is reclaimed with this command. For a serial tape device, the file names are changed to "*EMPTY", but no space is reclaimed. (Space on a serial tape device can be reclaimed by ZEROing all or part of the device.)

Since files are stored in a contiguous manner, unused spaces are left on the media when files are canceled. These unused spaces can be eliminated on directory-structured devices with the SQUISH command. SQUISH shifts the files together, leaving the free space in one contiguous area following the remaining files.

For a directory-structured device, the files specified must be CLOSEd when CANCEL executes. All files must be CLOSEd on a serial tape device when canceling files on it.

The CANCEL command does **not** issue a warning error if the specified file cannot be found on the device.

Using the Syntax Options:

The CANCEL command defaults to the system device if no device is specified. If the device specified is not the system device, its driver must be LOADed into memory before the command is executed. When the drive number is omitted, zero is assumed.

[If the peripheral is a serial tape, the **/F** or **/R** switch (Forward or Reverse) may be included to specify the direction of tape motion. Otherwise, the tape is rewound before a forward search begins. The switches are ignored with other devices. The switches are also ignored if a wild card specification (*) is used. Then, the entire tape is searched for matching file names.]

The wild card specification, an asterisk (*), can be inserted in place of either the file name, the extension, or both. If a file name is specified, and an asterisk appears in place of the extension, all files with that file name are canceled. Likewise, if the wild card appears in place of the file name, all files with the specified extension are deleted. Asterisks for both the file name and extension cancel all files on the device.

No default file name extension is provided with the CANCEL command.

CHAIN (Nonresident)

Examples:

```
100 CHAIN 'DECODE.SUB',1000
150 CHAIN R$,R
270 CHAIN CT:/F,"PART3"
```

Syntax Form:

$$[\text{line no.}]\ \textbf{CHAIN} \left[\text{device name[constant]:} \left[/ \begin{Bmatrix}\text{F}\\ \text{R}\end{Bmatrix} [,]\right]\right] [\text{string expression}][,\text{expression}]$$

Descriptive Form:

[line no.] **CHAIN** [device name[drive number]:[/forward or reverse switch[,]]]
[program file name][,line number where execution continues]

Purpose:

To delete all text and bring a new program into memory without disturbing variables.

Discussion:

This command is used to chain together segments of a large program. Unlike the OVERLAY command, CHAIN deletes all program text in memory before reading in the new program. The Scheduler stack and queue are cleared and the actions of all WHEN and SCHEDULE statements are canceled. However, like OVERLAY, CHAIN does not alter defined variables. That is, CHAIN is like OLD except OLD deletes both text and variables; CHAIN deletes text but not variables. CHAIN (as well as OVERLAY and OLD) does not CLOSE any OPEN files.

If the optional line number is present, execution continues at that line number in the new program. If that line does not exist in the new program, the first line with a higher line number is used. For example, in line 1ØØ above, execution would begin at line 1ØØØ in the program "DECODE.SUB" if there is a line 1ØØØ in "DECODE.SUB"; otherwise, execution begins at the first line whose number is greater than 1ØØØ.

[The new program executes with a task number equal to the task number of the CHAIN statement, unless that task number is 127 (the immediate mode task number). In that case, the task number is set to zero. Thus, the immediate mode command

```
CHAIN "NEXT",1
```

causes "NEXT" to execute as task number zero, not 127.]

If the line number is omitted, what happens depends on if the CHAIN command is issued in program mode or immediate mode. In immediate mode, the next command entered in immediate mode is executed. In program mode, execution continues with the first line of the new program. [Its task number is equal to the task number of the CHAIN statement, except when that task number is 127. Then the task number of the new program is zero.]

Since the optional line number in a CHAIN statement is an expression, it is **not** altered by the RENUM command.

Using the Syntax Options:

If no device is named, the program is assumed to be on the system device. If the named device is not the system device, its driver must be in memory when CHAIN is executed. (The keyboard, KB, may not be specified.) If no drive number is specified, zero is assumed.

[The Forward or Reverse switches (**/F** or **/R**) may be included in the command if the peripheral is a serial tape device. The switch specifies the direction of tape movement when searching for a file. If the switch is omitted, the tape is rewound before a forward search for the file is begun. The /F or /R switch is ignored when the device is not a serial tape device.]

A file name must be designated for a file-structured device. If no extension is present in the file name, .BAS is assumed.

If the optional expression is present, it is rounded to an integer and used as a line number. It must evaluate to an integer between 1 and 32767, inclusive. What happens when it is included or omitted is explained above.

CHANGE (Nonresident)

Examples:

```
100 CHANGE 100,"XY"
200 CHANGE "XY","ZZ"
    CHANGE 400,"OLD","NEW"
    CHANGE 240,500,A$,B$,DEL
```

Syntax Form:

[line no.] **CHANGE** [expression[,expression],]string expression[,string expression][,**DEL**]

Descriptive Form:

[line no.] **CHANGE** [line number[starting, line number ending],] text to be deleted [,text to be inserted][,**DEL**ete to end of line switch]

Purpose:

To alter or delete program text in memory.

Discussion:

The CHANGE command is used to edit program text. It can operate on any line of text in controller memory. Text is either altered or deleted by this command. When the command is executed in the immediate mode, the changed line or group of lines is printed on the terminal. When the CHANGE command is executed in program mode, the altered lines are not printed.

A CHANGE command may only appear as the first command in a line; it may not be preceded by a backslash (\).

If a RENUM command is executed on a program containing a CHANGE command, the expressions in the CHANGE command representing line numbers are **not** altered.

Using the Syntax Options:

Any expression appearing in the command is **truncated** to an integer value.

The optional first argument is the starting line number for the change and the optional second argument is the ending line number. If the second argument is not included in the command, only the first occurrence of the old text in the line specified by the first argument is altered. To change several occurrences of the old text in a single line, specify that line number as both the starting and ending line number.

When both line numbers are omitted, every occurrence of the old text in the entire program is altered.

The third argument is a string expression containing the text to be changed. Text in the specified range of line numbers matching this string is either deleted by the absence or changed by the presence of the optional fourth argument (also a string). If two strings are present, every instance of the third argument (the first string) found within the specified range of line numbers will be replaced by the fourth argument (the second string). If only one string appears, every instance of that string found within the specified range of line numbers will be deleted.

If the keyword **DEL** appears as the final argument, each time the old text is encountered within the specified range of text (and replaced or deleted), the remainder of the line in which it is found is deleted. That part of the line following the change and preceding the carriage return at the end of the line is deleted. Note that neither the carriage return nor any replacement text is deleted.

Application Examples:

The following examples demonstrate some variations of the CHANGE command:

```
1)   original text: 15Ø X=B+2.345+X
     command:       CHANGE 15Ø,"+2.3","+5.5"
     result:        15Ø X=B+5.545+X
```

```
2)  original text: 15Ø X=B+2.345+X
    command:       CHANGE 15Ø,"+2.3","+5.5",DEL
    result:        15Ø X=B+5.5

3)  original text: 15Ø REM THIS WAS A COMMENT
    command:       CHANGE 15Ø,"REM ",DEL
    result:        15Ø
```

Notice in example 3 that the text is deleted. If you were to type "LIST" following such a command, you would find line 15Ø gone. This illustrates how you can save controller memory if you are careful never to transfer control to a REM statement. After you OLD in a program, if you type:

```
CHANGE "REM ",DEL\RELEASE "CHANGE"
```

all REM statements are deleted from controller memory before you execute the program. Since both line numbers are omitted, all REM statements are deleted from the entire program. Be sure to include the space following the REM so that it is not mistaken for part of a word used elsewhere in the program. Failure to do so could produce unexpected results. A line like:

```
2ØØ PRINT "REMEMBER TO LOAD THE DRIVER"
```

becomes:

```
2ØØ PRINT "
```

CLEAR (Nonresident)

Example:

```
100 CLEAR
```

Syntax Form:

[line no.] **CLEAR**

Purpose:

To initialize all numeric variables to zero; all string variables to null.

Discussion:

CLEAR sets all defined variables (including arrays and waveforms) to zero. Strings are set equal to null strings. This command allows the user to initialize all the variables in a program to zero or null between successive runs of the same program.

CLOSE

Examples:

```
400 CLOSE #5
450 CLOSE #J*2
500 CLOSE ALL
```

Syntax Form:

$$\text{[line no.] } \mathbf{CLOSE} \begin{Bmatrix} \mathbf{\#}\text{expression} \\ \mathbf{ALL} \end{Bmatrix}$$

Descriptive Form:

$$\text{[line no.] } \mathbf{CLOSE} \begin{Bmatrix} \mathbf{\#}\text{plun} \\ \mathbf{ALL}\text{ pluns} \end{Bmatrix}$$

Purpose:

To close the data file or device currently associated with the specified peripheral logical unit number (PLUN) to further input or output.

Discussion:

Once a file has been CLOSEd, no further reference to that file can be made by input/output commands such as READ, PRINT or INPUT until it is OPENed again. The CLOSE statement releases the PLUN for use with other OPEN files. If no other PLUNs require it, the driver for the device can be RELEASEd (assuming it is not the system device) freeing the memory for another purpose.

When a line printer is CLOSEd, a form feed (skip to top of page) is output. When a paper punch is CLOSEd, a trailer (a blank length of tape on the end) is punched on the tape.

If the associated file or device is OPEN FOR READ, the CLOSE command merely dissociates the PLUN from it.

The CLOSE command has no effect if the specified PLUN is not currently OPEN.

The END statement also closes open files.

NOTE

A sequential-access file can only be reOPENed for READ. To add data to a CLOSEd sequential-access file, a new file must be OPENed for WRITE and the contents of the original file plus the new data written to it. However, a record I/O file can be reOPENed for UPDATE allowing access to it by the record I/O form of either the READU or WRITEU command.

Using the Syntax Options:

When an expression is supplied, it is evaluated and rounded to an integer. This integer is then used as the PLUN of the file or device to CLOSE. It must be between 1 and n, inclusive, where n is the number of PLUNs allowed at initialization (default of six). It is illegal to attempt to CLOSE the keyboard (PLUN zero).

If the keyword **ALL** is specified instead of a PLUN, all files and devices currently OPEN are CLOSEd to further input or output. Each is individually CLOSEd according to the rules discussed earlier.

COPY (Nonresident)

Examples:

```
670 COPY CT1:/F,"PROG.LIST" TO LP:
    COPY A$ TO DX1:B$
    COPY "BASIC.DAT" TO KB:
    COPY DK1:"FILE.*" TO PP:
    COPY F$&".*" TO DX1:F$&".*"
    COPY DX1:"FILE.DAT" TO "TEST.DAT" INTO 5
    COPY DX:"*.SPS" TO DK:"*.SPS"
```

Syntax Form:

```
                                          ⎡  ⎧F⎫     ⎤
[line no.] COPY [device name[constant]:]  ⎢/ ⎨ ⎬ [,] ⎥ [string expression]
                                          ⎣  ⎩R⎭     ⎦
                TO [device name[constant]:][string expression][ INTO expression]
```

Descriptive Form:

```
[line no.] COPY [device name[drive number]:][/forward or reverse switch[,]]
                [source file name] TO [device name[drive number]:][target file name]
                [ INTO number of blocks].
```

Purpose:

To transfer data from one peripheral device directly to another, or to make a second copy of a file on a single peripheral.

Discussion:

This command provides a convenient means of transferring programs or data from one device to another or of creating additional copies of a file on the same peripheral.

Transfers between files on the same peripheral are legal only if two files may be OPEN simultaneously on that peripheral.

There must be an unused peripheral logical unit number (PLUN) available when the COPY command is executed because the COPY command temporarily OPENs the source file. If a Control-P is typed while the COPY command is executing, this PLUN is left open, preventing future use of that PLUN. Also, if an error occurs during the COPY command, the source file might be left OPEN. By executing STATUS you can determine which PLUN is associated with the source file and then CLOSE that PLUN. Or if you prefer, you can enter a CLOSE ALL or END instead of using STATUS and explicitly closing that one file. However, this second method closes all OPEN files.

When COPYing data from the keyboard (KB is the source device), pressing the Return key outputs only a carriage return -- not the usual carriage return and line feed. This means that you can use the keyboard to COPY ASCII data directly to a file. The data -- ASCII strings -- will be terminated (delimited) by a carriage return each time you press the Return key. Such data files can be read by the INPUT command. To terminate the COPY command when the keyboard is the source device, enter a Control-Z.

Using the Syntax Options:

The first device and file name specified is the source. The second device and file name specified is the target. The keyword **TO** separates the two. If the target device is file-structured, no file may already exist on it with the same name as the target file specified in the COPY command. If the target device is not file-structured, a file name need not appear with that device. It must be legal to write to the target device.

The system device is the default device for the COPY command. If the device specified is not the system device or the keyboard (KB), its driver must be LOADed into memory before the command is executed. When the drive number is omitted, zero is assumed.

If the terminal keyboard (KB) is the source device, a question mark (**?**) is printed when the system is ready to accept input. Enter your input after the question mark (**?**) and terminate the input by entering a Control-Z.

[The Forward or Reverse switches (**/F** or **/R**) are used only if the source device is a serial tape device. These switches specify the direction of tape movement when searching for the source file. If the switch is omitted, the tape is rewound before a forward search for this file begins. For other peripherals, these switches are ignored. These switches are also ignored if a wild card specification (*****) is used.]

A wild card specification can be used in place of the file name, the extension, or both. The wild card specification is indicated by an asterisk (*). If the source extension is given as an asterisk, each file with the specified name, regardless of extension, is transferred to the destination. The file name may also be replaced by an asterisk. This causes all files with the specified extension to be transferred. If the source has a wild card in either the file name or extension, the destination must have a wild card in the corresponding position, or be a device which doesn't require a file name, such as line printer (LP) or the keyboard (KB).

CAUTION

If a wild card specification (*) is used in the source file name and a serial tape device which has more than one file with that given name is the source device, the command may not function predictably. The data from the wrong file may be transferred.

There is no default file name extension provided with the COPY command.

If the target device is directory-structured, the **INTO** option can be used. The expression following the keyword INTO stipulates the maximum number of blocks required by the file being copied. The first sufficient empty space on the target device is selected for the file. When the INTO option is not used, one half of the largest empty space on the target device is opened for the file. In either case if the specified or default space exceeds the actual number of blocks required by the file, the unused blocks are returned to an empty status.

Use of the INTO option with the wild card (*) notation is unnecessary. The block number stipulation is ignored and as each file is transferred, the first sufficient empty space on the target device is used for that file.

When COPYing to a nearly full disk, use the INTO option or the wild card notation (*). Half the remaining free space may not be large enough for the file.

DATE (Nonresident)

Examples:

```
150 DATE A$
260 DATE A(Ø:2)
185 DATE D
    DATE
```

Syntax Form:

```
                     ┌simple numeric variable┐
[line no.] DATE      │array                  │
                     └string variable        ┘
```

Descriptive Form:

```
                     ┌target variable        ┐
[line no.] DATE      │target array           │
                     └target string variable ┘
```

Purpose:

To return the system date.

Discussion:

The DATE command either returns the system date in the specified argument or prints the date on the terminal if the argument is omitted. When the argument is supplied, the data is returned either as three array elements or a string, depending on the type of variable specified.

When the date is returned as three array elements, they are stored in the array in this order:

first element	month (1-12)
second element	day of month (1-31)
third element	year (72-99)

When the date is returned in a string variable, it is of the form:

DD-MMM-YY

where:

DD	day of month (1-31)
MMM	first three letters in name of month
YY	year (72-99)

The system date is set by the SETDATE command. When the system is booted, the date is cleared. Also, since the date is not automatically updated, it should be reset each day.

Using the Syntax Options:

Specifying either a simple (not subscripted) variable or an array returns the date in an array. If a simple numeric variable is used, it is auto-dimensioned to a three-element integer array. If an array is used, it must be dimensioned or zoned to three elements.

Specifying a string variable returns the date in that string.

Omitting the argument prints the date on the terminal in the string variable format.

Application Example:

The DATE command can be used to print the date on program runs. A simple method is to return the date as a string and print it. For example:

```
1ØØ DATE D$
11Ø PRINT #N,"RUNDATE: ";D$
```

where N is assumed to be the peripheral logical unit number (PLUN) of a line printer that is OPEN FOR WRITE. To output the date in a different format, such as "MMM DD, 19YY", you could use the SEG function:

```
        12Ø PRINT #N,"RUNDATE: ";SEG(D$,4,6);" ";SEG(D$,1,2);",19";SEG(D$,8,
```

DEFINE (Nonresident)

Examples:

```
DEFINE DX1:'RECORD.DAT' AS ARR 1Ø,VAR,STG 2 WITH 1ØØ
DEFINE A$ AS ARR X*7,STG Z,IAR X WITH A*B/2
DEFINE 'TEST.DAT' AS VAR,VAR,STG 1Ø WITH 1Ø
```

Syntax Form:

```
[line no.] DEFINE [device name[constant]:]string expression
                  ⎧VAR            ⎫ ⎡  ⎧VAR            ⎫⎤
               AS ⎨ARR expression ⎬ ⎢, ⎨ARR expression ⎬⎥ ... WITH expression
                  ⎨IAR expression ⎬ ⎢  ⎨IAR expression ⎬⎥
                  ⎩STG expression ⎭ ⎣  ⎩STG expression ⎭⎦
```

Descriptive Form:

```
[line no.] DEFINE [device name[drive number]:] file name
                  ⎧VARiable                                    ⎫
               AS ⎨ARRay number of floating-point elements     ⎬
                  ⎨Integer ARray number of integer elements    ⎬
                  ⎩STrinG number of characters in string       ⎭

               ⎡  ⎧VARiable                                    ⎫⎤
               ⎢, ⎨ARRay number of floating-point elements     ⎬⎥ ...
               ⎢  ⎨Integer ARray number of integer elements    ⎬⎥
               ⎣  ⎩STrinG number of characters in string       ⎭⎦

               WITH number of records
```

Purpose:

To allot space for a record I/O file on the specified directory-structured device.

Discussion:

A record I/O (Input/Output) file is a data file with data can be accessed randomly -- any logical record at a time -- in order to enter, retrieve, or update a data record. In this sense, a data record is a set of related items of data treated as a unit; all the records are the same length. Instead of being OPENed for either READ or WRITE, a record I/O file is OPENed FOR UPDATE, which allows both input and output operations.

The first step in using a record I/O file is to create a file of sufficient length on a directory-structured peripheral device. The DEFINE command does this and even makes it unnecessary for you to count the number of words or bytes required. You need only describe (with keywords) the contents of the data record and the number of records desired. The command determines the size of the file by computing the number of bytes per data record and multiplying this by the number of requested records. As the command creates the file on the peripheral, the file is zeroed.

Once the file is created, it can only be written to in TEK SPS BASIC by a special form of the WRITEU command. [Since the WRITEU command outputs data to a file in an unformatted binary form, there are no data descriptors and no logical end-of-record markers written on the file. This gives you the flexibility to logically restructure the file when accessed.]

See the OPEN, WRITEU, READU, and CLOSE commands for related discussions.

Using the Syntax Options:

The device name is the peripheral on which the file is generated. Unless this peripheral is a directory-structured device such as a hard or floppy disk, a fatal error results. If no device is specified, the system device is used. If the named device is not the system device, its driver must be in memory when the command is executed. If no drive number is supplied, zero is assumed.

A file name is required. It must not be the name of a file that already exists on the peripheral. A fatal error is issued if it is.

The contents of a data record are described with the keywords ARR, IAR, VAR, and STG. **ARR** describes a floating-point array with each element four bytes long; **IAR** describes an integer array with each element two bytes long. The experssion following ARR or IAR is the <u>number of elements</u> in the array (not the dimension, but the <u>size</u> of the array). **VAR** describes a single, floating-point variable, while **STG** describes a string variable. The expression following STG is the number of characters, and therefore the number of bytes, in the string. (The keyword IAR is not supported by DEFINE VØ2-Ø1.)

The total number of bytes in a logical record is calculated from the keyword information. This record length is then multiplied by the number of records requested in the expression following the keyword **WITH**. This product determines the minimum size of the file. [The actual size of the file must be an integral multiple of a block (256 words). Thus, a file, whose calculated size is 6ØØ bytes (3ØØ words), is really two blocks long. Since there are no logical end-of-file markers on record I/O files, the entire file space is accessible. Any room between the logical end-of-file and the physical end-of-file may be used for additional data. The physical end-of-file is the physical end of the last block.]

All numeric expressions are rounded to integers.

Application Example:

Let's look at a few examples. Suppose you wanted a record I/O file named FLOPPY.IO on a floppy disk. Each record is to contain a string variable 2Ø characters long, a floating-point variable, a floating-point array of 512 elements, another floating-point variable, and finally two more string variables of 1Ø characters each. There are to be 25 of these records. The following DEFINE statement does this for you.

```
DEFINE DX1:"FLOPPY.IO" AS STG 2Ø,VAR,ARR 512,VAR,STG 1Ø,STG 1Ø WITH 25
```

[But since the DEFINE command only creates and zeroes a file and does not section the file, you could also use:

```
DEFINE DX1:"FLOPPY.IO" AS STG 4Ø,ARR 512,ARR 2 WITH 25
```

or even just:

```
DEFINE DX1:"FLOPPY.IO" AS STG 4Ø,ARR 514 WITH 25
```

to allot space for the same file. Notice, in the latter two examples, how the string variables are collected under STG and the numeric variables under ARR.]

DELETE

Examples:

```
150 DELETE C$,677
    DELETE ALL
    DELETE TEXT,A,DD
575 DELETE 250,300,400
```

Syntax Form:

$$\text{[line no.] }\textbf{DELETE}\left\{\begin{array}{l}\left\{\begin{array}{l}\text{array}\\\text{waveform}\\\text{string array}\\\text{line number[,line number]}\\\textbf{TEXT}\end{array}\right\}\left[\text{, }\left\{\begin{array}{l}\text{array}\\\text{waveform}\\\text{string array}\\\text{line number[,line number]}\\\textbf{TEXT}\end{array}\right\}\right]\ldots\\\textbf{ALL}\end{array}\right\}$$

Descriptive Form:

$$\text{[line no.] }\textbf{DELETE}\left\{\begin{array}{l}\left\{\begin{array}{l}\text{array}\\\text{waveform}\\\text{string array}\\\text{line number[starting,line number ending]}\\\text{all program }\textbf{TEXT}\text{ in memory}\end{array}\right\}\\\textbf{ALL}\text{ program text and data in memory}\end{array}\right.$$

$$\left.\left[\text{, }\left\{\begin{array}{l}\text{array}\\\text{waveform}\\\text{string array}\\\text{line number[starting,line number ending]}\\\text{all program }\textbf{TEXT}\text{ in memory}\end{array}\right\}\right]\ldots\right\}$$

Purpose:

To remove arrays, string arrays, waveforms, or program lines from memory.

Discussion:

This command frees memory space by deleting defined arrays or sections of program text. Deleting an array changes it to a simple floating-point or string variable. Thus, deleting an array allows you to redimension it to different specifications. Once deleted, the arrays or program lines are not recoverable. Only the **DELETE ALL** form of the command removes simple floating-point or string variables.

Since a DELETE ALL removes all program text from memory, a DELETE ALL statement should not be followed by a backslash (\).

Using the Syntax Options:

An array is removed by explicitly naming it. If a waveform is specified, the array, data sampling interval, and units strings are dissociated from each other. But, the array is not deleted unless you specifically name the array.

Line numbers are treated as single lines or sequential groups. If one line number is given, that line is removed from memory. If two line numbers appear in the DELETE statement in sequence, all lines within their range (inclusive) are removed. Other combinations of line numbers delete either one line, or a sequence of lines. Let's look at some detailed examples:

```
100 DELETE 50,200,600
200 DELETE 50,A,300,600
300 DELETE 50,200,600,700
```

Line 100 deletes all text between lines 50 and 200, inclusive, and also line 600. The statement at 200 deletes line 50, array A, and all lines between 300 and 600, inclusive. Line 300 deletes all lines between 50 and 200 and all lines between 600 and 700, inclusive.

The keyword **TEXT** removes <u>all</u> program statements, but does not alter any variables. The actions of all WHEN, SCHEDULE, INPREQ, and ONERR statements are canceled and the Scheduler's stack and queue are cleared.

The keyword **ALL** removes <u>all</u> program lines and <u>all</u> variables from memory. The actions of all WHEN, SCHEDULE, INPREQ, and ONERR statements are canceled and the Scheduler's stack and queue are cleared.

Uses:

DELETE can save you work. Suppose you want to use the same section of program code in a new program that you have in a SAVEd program. OLD in the program and DELETE the lines you don't wish to keep. Then, if necessary RENUMber the section of code you retained before you start adding the new code.

DETACH

Examples:

```
160 DETACH #1
170 DETACH #G
    DETACH ALL
```

Syntax Form:

$$[\text{line no.}]\ \textbf{DETACH}\ \left\{\begin{array}{l}\textbf{\#}\text{expression}\\ \textbf{ALL}\end{array}\right\}$$

Descriptive Form:

$$[\text{line no.}]\ \textbf{DETACH}\ \left\{\begin{array}{l}\textbf{\#}\text{ilun}\\ \textbf{ALL}\ \text{iluns}\end{array}\right\}$$

Purpose:

To terminate communication with an instrument by dissociating an instrument logical unit number (ILUN) from its associated instrument.

Discussion:

This command is the opposite of the ATTACH command. It dissolves the logical connection between an instrument and an instrument logical unit number (ILUN). When an ILUN is DETACHed, that ILUN is freed for use with another instrument. The instrument is effectively no longer on line, and no communication can take place with it until the instrument is again ATTACHed.

DETACH issues no error if the ILUN is not ATTACHed.

Using the Syntax Options:

When an expression is given, it is rounded to an integer and used as an ILUN. It must evaluate to a number between 1 and n, inclusive, where n is the number of ILUNs requested at initialization (default of eight). Only that ILUN is DETACHed.

When the Keyword **ALL** is used instead of an ILUN, every ATTACHed instrument is dissociated from its corresponding ILUN.

DIM

Examples:

```
10 DIM X(99)
20 DIM P(511),R(A,B),X$(15)
```

Syntax Form:

$$\text{[line no.] }\textbf{DIM}\left\{\begin{array}{l}\text{simple numeric variable}\\\text{simple string variable}\\\text{floating-point array}\\\text{string array}\end{array}\right\}\textbf{(}\text{expression[}\textbf{,}\text{expression]}\textbf{)}$$

$$\left[\textbf{,}\left\{\begin{array}{l}\text{simple numeric variable}\\\text{simple string variable}\\\text{floating-point array}\\\text{string array}\end{array}\right\}\textbf{(}\text{expression[}\textbf{,}\text{expression]}\textbf{)}\right]\ldots$$

Descriptive Form:

$$\text{[line no.] }\textbf{DIM}\left\{\begin{array}{l}\text{simple numeric variable}\\\text{simple string variable}\\\text{floating-point array}\\\text{string array}\end{array}\right\}\textbf{(}\text{first dimension [}\textbf{,}\text{second dimension]}\textbf{)}$$

$$\left[\textbf{,}\left\{\begin{array}{l}\text{simple numeric variable}\\\text{simple string variable}\\\text{floating-point array}\\\text{string array}\end{array}\right\}\textbf{(}\text{first dimension[}\textbf{,}\text{second dimension]}\textbf{)}\right]\ldots$$

Purpose:

To allocate storage space for floating-point arrays and string arrays.

Discussion:

An array is a set of variables that are stored (contiguously) under the same name. Each element is referenced by the array name and its index. In TEK SPS BASIC, arrays can have one or two dimensions. That is, they can be thought of as a single column of elements or as a matrix with rows and columns.

The array indices are numbered from zero. An array A of DIMension N has N+1 elements. The first element in A is A(Ø). The last element is A(N). Similarly, a matrix B of DIMension I,J has I+1 by J+1 elements. The first element in B is B(Ø,Ø) while the last element is B(I,J).

The DIMension command reserves memory space for floating-point (numeric) or string arrays. The space is allocated as the DIM statement is encountered. In standard memory systems, the size of a floating-point array is limited only by the amount of available free memory. In extended memory (XM) systems, the maximum size of a floating-point array is limited to 8192 (8K) elements. If there is not enough memory available to contain the array, a fatal error is issued.

When a floating-point array is specified, two words of controller memory are required for each element. Thus a floating-point array of N elements needs 2*N words of memory. (To dimension an integer array which requires only one word of memory per element, use the INTEGER command.)

The amount of memory needed for a string array cannot readily be determined. When a string array is dimensioned, it initially requires one word for each element. However, each string in the array can be of differing length and can grow or shrink in length during program execution. In other words, string length is dynamic. Thus, the amount of memory space required for a string array depends on the length of each string, not just on the number of elements in the array.

Auto-dimensioning:

Arrays may also be automatically dimensioned during program execution of commands such as LET and READ. With such commands, if a simple numeric variable is the destination of an expression which results in an array, the simple numeric variable is dimensioned to the size of the source array.

Using the Syntax Options:

The numeric or string variable is the name of the array. Specifying a simple (not subscripted) numeric variable allocates space for a floating-point array. Using a string variable assigns storage for a string array. In either case, if the specified variable is already an array, its dimension(s) must not be changed. To redimension an array to new specification you must DELETE the array first.

The numeric argument in parentheses determines the number of dimensions (one or two) and the size of the array. An expression is rounded to an integer and used as the largest index -- not the number of elements -- in a row or column. Specifying a single argument that evaluates to the integer N allocates a one-dimensional array of N+1 elements. Supplying two arguments creates a two-dimensional array. If the expressions evaluate to the integers, I and J, space is allocated for a matrix of I+1 by J+1 elements.

DIR (Nonresident)

Examples:

```
150 DIR
    DIR WITH BLOCK
160 DIR DX1: TO DX1:"DIRFIL"
    DIR DK2:"*.BAS" TO LP:
    DIR EXC
```

Syntax Form:

```
[line no.] DIR [EXC [device name[constant]:]                          ]
               [[device name[constant]:][string expression]           ]
               [WITH BLOCK] [TO [device name[constant]:][string expression]]
```

Descriptive Form:

```
[line no.] DIR [EXClude .SPS files [device name[drive number]:]                              ]
               [[device name[drive number]:] [file name or wild card specification]           ]
               [WITH starting BLOCK numbers printed] [TO [device name[drive number]:]
               [file name to receive directory information]]
```

Purpose:

To send a listing of the directory of the files stored on a device to the terminal, another device, or a file.

Discussion:

The DIRectory command lists the names of the files stored on a file-structured device. The listing can be sent to the terminal, a device such as a line printer, or a file. The names of all or part of the stored files can be selected. For each file listed, the name, size, and creation date of the file is printed. For a block-structured source device, the starting block numbers of the files can also be printed. When the source device is a directory-structured device, the listing includes the unused blocks and the number of free blocks on the device.

Using the Syntax Options:

Everything except the command name is optional. Any specification to the left of the keyword **TO** describes the source. Any specification to the right of the TO describes the destination device or file. Omitting all source specifications lists the directory of the system disk. Omitting the keyword TO and all the associated destination specifications, sends the listing to the terminal.

Specifying the keyword **EXC** lists all files on the source device except those files with the .SPS extension. (The EXC option is not supported by DIR VØ2-Ø1.)

The source device must be file-structured. If no device is named, the system device is assumed. If the device is not the system device, its driver must be in memory before the DIR command is executed. If no drive number is included, zero is assumed.

When a file name is included in the source, only that file's information (plus the list of empty blocks for a block-structured device) is listed. Usually, though, the file name has a wild card asterisk (*) specification in it in order to print a part of the directory. If an asterisk is used in the name portion of the file name, every file with the specified extension is printed. If an asterisk appears as the extension, every file with the specified name, regardless of extension, is printed. Using an asterisk in place of both the name and the extension has the same effect as omitting the file name: the entire device directory is printed.

[If the optional **WITH BLOCK** keywords are present, the starting block numbers (in octal) of the files are also printed. If the device is not block-structured, a warning error is issued and the WITH BLOCK directive is ignored.]

The optional keyword **TO** is used to send the listing to some device other than the terminal. The device following the TO becomes the destination device. If this device is omitted, output goes to the terminal. The driver for the destination device must be in memory before the DIR executes. If the drive number is omitted, zero is used.

A file name must be supplied if the destination device is file-structured. A file with the same name must not already exist on the device.

END

Example:

```
199 END
```

Syntax Form:

[line no.] **END**

Purpose:

To end program execution and return BASIC to idle mode.

Discussion:

The END command terminates a running program. It clears the Scheduler stack and queue of all tasks, returning the monitor to idle mode. END cancels the action of all WHEN statements and clears the clock queue. It also disables any INPREQ or ONERR command conditions and CLOSEs any OPEN files. However, any ATTACHed instruments remain ATTACHed. (Notice that, unlike STOP, END CLOSEs files.)

END may appear anywhere in a program and may even be omitted from a program.

Since it clears the Scheduler, executing END halts all tasks, not just the one in which it appears. To halt the current task and only that task, use an ABORT statement with the task number omitted. This terminates only the currently executing task. (ABORT does not CLOSE files. If you need to free peripheral logical unit numbers (PLUNs), also use an appropriate CLOSE statement.) To terminate the program, yet leave the Scheduler intact for processing pending interrupts, use RETURN instead of END. (The function and parts of the Schedules are explained in Section 1.) .

EOF (Nonresident)

Examples:

```
100 EOF #3 GOTO 3300
246 EOF #A*2 GOTO 25699
```

Syntax Form:

[line no.] **EOF #**expression **GOTO** line number

Descriptive Form:

[line no.] **EOF #**plun **GOTO** line number

Purpose:

To designate a program line to receive control when data from a peripheral file is exhausted.

Discussion:

Normally, when you attempt to read beyond the end of a file, a fatal error is issued and the program (task) stops. However, you may not know beforehand the length of a file. In such a case, the EOF (End Of File) command may be used. After this statement has been executed, an attempt to read past the end of a file causes program control to be transferred to the line number specified in the command.

More than one EOF command per file may be executed. The last EOF command executed determines the line number to which control is transferred when the file is exhausted.

Using the Syntax Options:

The peripheral logical unit number (PLUN) specified must be a file OPENed FOR READ or UPDATE before the EOF command can be executed. The keyboard (PLUN zero) may not be used.

Application Example:

As an example, the following program reads an unknown number of strings from a file, and displays the number of strings read.

```
100 OPEN #3 AS DK2:"STRING.FIL" FOR READ
110 EOF #3 GOTO 900
120 C=0
130 INPUT #3,A$
140 C=C+1
150 GOTO 130
900 PRINT "NUMBER OF RECORDS READ IS";C
910 CLOSE #3
```

In this sample program, line 100 prepares a file called STRING.FIL on disk drive two to be read. The file is assigned to PLUN 3. From then on, only #3 need be typed to access this file.

Line 110 instructs BASIC to jump to line 900 when the file has been completely read. Line 120 sets the string counter to zero. The variable C is used to keep track of the number of strings in the file.

Line 130 does the reading. The INPUT statement reads one string from the file, and puts it into string A$. (This program assumes the strings were output by a PRINT command.) Statement 140 increments the string counter by one, thus counting each string as it is read. Statement 150 directs BASIC back to line 130 to read another string.

When all strings have been read, program control automatically jumps to line 900, as directed in the EOF statement. Here, at line 900, the PRINT statement displays the message "NUMBER OF RECORDS READ IS" and the value of C. Line 910 closes the file, and dissociates PLUN 3 from the file.

FOR

Examples:

```
10 FOR I = 0 TO 100
20 FOR P = -9 TO 10 STEP .5
30 FOR Q = X TO Z STEP -N
```

Syntax Form:

[line no.] **FOR** simple numeric variable = expression **TO** expression[**STEP** expression]

Descriptive Form:

[line no.] **FOR** index = initial value **TO** limit [**STEP** increment]

Purpose:

To provide a program loop when paired with a matching NEXT statement.

Discussion:

The FOR command, paired with a matching NEXT command, forms a program control loop. The lines within the loop are repeated as many times as the FOR statement parameters define, using the following algorithm:

When a FOR command is encountered, the parameter expressions are evaluated and the index variable is set to the initial value (the first expression). Control then passes to the command following the FOR command. When the matching NEXT statement (the NEXT statement having the same index variable) is executed, the increment value (default value of one) is added to the index. The new index value is compared with the loop limit value (the second expression in the FOR statement). If the new index value is less than or equal to the limit (or greater than or equal to the limit if the increment value is negative), control passes back to the command immediately <u>following</u> the matching FOR statement. The loop repeats in this manner until the index is greater than (or less than if the increment is negative) the limit value.

The loop always executes at least once, no matter what its parameters are.

The expressions in the FOR statement are evaluated only on the first pass through the loop. Thus if variables are used to define the increment and limit values of the loop, alteration of these variables within the loop will not change the range of the index variable. The index variable itself may be altered, however, to change the duration of the loop.

Negative steps are allowed in FOR loops. In this case, the initial value of the index variable should be greater than the limit value. If it is not, the loop executes only once.

For each FOR command in a BASIC program, there must be a matching NEXT command. Loops may be nested (more than one FOR loop in progress at a time) but an inner loop must be contained completely within an outer loop. Examples of legal and illegal FOR/NEXT loops follow.

```
        LEGAL                              ILLEGAL

10 FOR I = 1 TO 5                   10 FOR I = 1 TO 5
20 FOR J = 7 TO 1 STEP -1           20 FOR J = 7 TO 1 STEP -1
30 NEXT J                           30 NEXT I
40 NEXT I                           40 NEXT J
```

In the legal example, the inner J loop is completely contained within the outer I loop. In the illegal example, the J loop extends outside the outer I loop.

Two notes of caution are in order here. When very large numbers are incremented by very small numbers, unexpected results can occur. For example, the following FOR loop will never terminate.

```
FOR I = 1000000 TO 1000001 STEP .01
```

This is because the step value (.01) is insignificant in relation to the range values. Also, because of binary number limitations, when the increment value is a fraction, the loop may not repeat as often as you might expect. For more information about binary number limitations, see Section 2 of this manual.

In order for a FOR/NEXT loop to execute in immediate mode,the entire loop must be concatenated on one line. For example:

```
FOR L = Ø TO 9\PRINT M$(L)\NEXT L
```

prints the first ten items of string array M$.

NOTE

Use care if you include within a FOR/NEXT loop any statements such as those that OVERLAY, CHANGE, or DELETE lines of program text. If the actions of a FOR/NEXT loop modify program text, any line that is added, overlaid, altered, or deleted must not have a line number smaller than that of the FOR statement. A fatal error results if it does.

Using the Syntax Options:

The simple numeric variable is the loop index. The FOR command and its matching NEXT command must use the same variable name for the loop index.

The first expression is the initial index figure -- the value of the index the first time through the loop.

The second expression is the limit value -- the number to which the index is compared.

The optional expression following the keyword **STEP** is the increment value, the number added to the index at the end of each pass through the loop. The increment may be negative if the initial value is greater than the limit. When the increment is omitted, the step size is assumed to have a value of one.

Application Example:

The following is a classic "bubble sort" routine that arranges the numbers in array A in ascending order:

```
100 REM NUMERIC BUBBLE SORT
110 FOR I=1 TO SIZ(A)-1
120 IF A(I)>=A(I-1) THEN 190
130 FOR J=I TO 1 STEP -1
140 IF A(J)>=A(J-1) THEN 190
150 T=A(J)
160 A(J)=A(J-1)
170 A(J-1)=T
180 NEXT J
190 NEXT I
```

The outer loop searches down through the array looking for a number out of order, an A(I) less than an A(I-1). When one is found, its correct place in the part of the array already sorted is sought. The inner loop is used to "bubble" the A(I) value up through the sorted values until it is in its correct place. A variable, T, temporarily holds the value of each A(J) when the values of A(J) and A(J-1) are switched.

The bubble sort is a very slow sort routine algorithm. For a fast sort routine, see the example shown in the minimum (MIN) function discussion.

FORMAT (Nonresident)

Examples:

```
FORMAT DK1:
FORMAT DY:VER
FORMAT DK2:1Ø
FORMAT DK1:2Ø,VER
FORMAT DY1:10,SINGLE,VER
```

Syntax Form:

```
                          ┌expression[,SINGLE][,VER]┐
[line no.] FORMAT {DK} [constant]:│SINGLE[,VER]             │
                  {DY}            └VER                      ┘
```

Descriptive Form:

```
                                   ┌number of directory segments [,SINGLE density]┐
                  {DK}                                           [,VERify]        │
[line no.] FORMAT {DY} [drive number]:│SINGLE density [,VERify]                      │
                                   └VERify                                        ┘
```

Purpose:

To format either a DEC RK05 (or equivalent) hard disk or a DEC RX02 (or equivalent) dual-density floppy disk.

Discussion:

Each disk device driver expects the disk to be formatted in a prescribed manner (e.g., each sector is expected to contain a particular header followed by a data space of set size.) These prescribed formats are discussed in the Peripheral Drivers manual. Although most disks are factory-formatted, the FORMAT command allows you to format two types of disks: the DEC RK05 (or equivalent) hard disk and the DEC RX02 (or equivalent) dual-density floppy disk.

The FORMAT command also allocates room for the device directory and initializes the disk, logically zeroing the directory and data areas. This means that a disk which has been formatted by the FORMAT command does not need to be initialized by the ZERO command before it is used for the very first time. However, factory-formatted disks do need to be initialized by ZERO.

CAUTION

The FORMAT command is intended for use on a blank disk. It initializes the disk after formatting it, so any data on the disk is effectively erased and cannot be recovered.

Using the Syntax Options:

This command only formats disks that use the DK Hard Disk driver (DK.SPS) or the DY Dual-Density Floppy Disk driver (DY.SPS). The appropriate driver must be in memory when FORMAT executes. If the drive number is omitted, the disk in drive Ø is formatted.

NOTE

The DY driver is not available in TEK SPS BASIC VØ2-Ø1. Also, FORMAT VØ2-Ø1 does not format an RX02 (or equivalent) disk.

The optional expression determines the number of directory segments allocated. The expression, when evaluated and rounded to an integer, must be between 1 and 31, inclusive. If this number is omitted, a default number of directory segments are allocated. The default is 8 for the DK driver and 4 for the DY driver.

The space allotted for the directory must be large enough to hold the names of all the files to be stored on the disk. If most of the files are large (ten blocks or more), the default value may suffice. However, if most of the files are small (about two blocks in length), you may need several times more than the default number of directory segments. For more guidance on how many directory segments to allocate, see the Peripheral Drivers manual.

If the device is DY, the disk can be formatted for either double-or single-density data storage. Specifying the keyword **SINGLE** formats the disk for single-density. Omitting it formats the disk for double-density. The keyword is ignored if used with DK.

The command can also verify the disk. When the optional keyword **VER** is used, the disk is checked for bad blocks <u>after the formatting is done</u>. If any bad blocks are found, their block numbers (in octal) are printed on the terminal. (Even if bad blocks are found, the disk will still be initialized, but a P18 error is issued after the command finishes executing.)

As an example, the single statement:

```
FORMAT DK1:15,VER
```

formats the hard disk in drive 1, allocates 15 directory segments, checks the disk for bad blocks, and initializes it.

GET (Nonresident)

Examples:

```
150 GET A1 FROM #3,A$
160 GET B$,C$ FROM #17,"SCAN","GRAT"
170 GET #1 FROM #J,"FAS"
180 GET #3 FROM #1
190 GET A$ FROM @0,TA,SA
240 GET AA$ FROM #3;4,"SET?"
```

Syntax Form:

```
                  ⎧#expression                                                        ⎫
                  ⎪ ⎧variable        ⎫ ⎡    ⎧variable        ⎫⎤   ⎪
                  ⎪ ⎪array           ⎪ ⎢    ⎪array           ⎪⎥   ⎪
[line no.] GET    ⎨ ⎨waveform        ⎬ ⎢{,} ⎨waveform        ⎬⎥...⎬
                  ⎪ ⎪string variable ⎪ ⎢{;} ⎪string variable ⎪⎥   ⎪
                  ⎩ ⎩string array    ⎭ ⎣    ⎩string array    ⎭⎦   ⎭
                  FROM {#expression[;expression][,string expression] ...}
                       {@expression,expression[,expression]             }
```

Descriptive Form:

```
                  ⎧#target plun to receive data                                                 ⎫
                  ⎪ ⎧target variable        ⎫ ⎡    ⎧target variable        ⎫⎤   ⎪
                  ⎪ ⎪target array           ⎪ ⎢    ⎪target array           ⎪⎥   ⎪
[line no.] GET    ⎨ ⎨target waveform        ⎬ ⎢{,} ⎨target waveform        ⎬⎥...⎬
                  ⎪ ⎪target string variable ⎪ ⎢{;} ⎪target string variable ⎪⎥   ⎪
                  ⎩ ⎩target string array    ⎭ ⎣    ⎩target string array    ⎭⎦   ⎭
                       ⎧#source ilun [;secondary address]                                    ⎫
                       ⎪              [,driver-dependent specification of data or status   ⎪
                  FROM ⎨               information to be obtained from instrument]...      ⎬
                       ⎩@IEEE 488 interface number, talk address [,secondary address]      ⎭
```

Purpose:

To acquire data or status information from a specified instrument.

Discussion:

The GET command fetches data or status information from an instrument and stores it in variables in memory, or sends it directly to a peripheral storage device.

The GET command is divided into two parts: the target and the source. The target may be a single peripheral logical unit number (PLUN) or a list of variables. If the target is a PLUN, data acquired from the instrument is sent directly to the peripheral device. This method of acquisition is known as "data-logging" and allows very rapid data acquisition. Not all instrument drivers support data-logging. When used, only one PLUN can appear in a GET statement. The PLUN must be OPEN for WRITE at the time the GET statement is executed.

If the target variable is a simple numeric variable and the GET acquires a waveform or array, depending on the instrument driver, the target variable may be auto-dimensioned to an array of the appropriate length and type.

The second portion of the command indicates the source instrument by either the instrument logical unit number (ILUN) of the ATTACHed instrument or the IEEE 488 interface number followed by an address. If the source is indicated by an ILUN, one or more source strings may follow to communicate driver-dependent information. Usually a one-to-one relationship exists between a source string and a target variable. Each instrument driver recognizes a different set of strings. For any instrument, only those strings that its driver responds to should be used. Complete documentation of the driver-dependent strings that a driver responds to can be found in the manual for the specific instrument driver used.

When the GET command executes, the instrument must be on-line and the required instrument driver must be LOADed in memory. Also, either the instrument must be ATTACHed to associate it with the instrument logical unit number (ILUN), or the communication must be through the low-level IEEE 488 Interface driver, GPI.SPS, which is discussed in Section 6.

Using the Syntax Options:

No instrument driver uses all the legal syntax variations of the GET command. The manual for each driver shows which of the forms are allowed by that driver.

If the target is an expression following a pound sign (#), a peripheral logical unit number (PLUN) is specified and the acquired data will be sent directly to that peripheral.

If a list of one or more target variables is used, the data or information is stored in the controller memory. The list may include numeric variables, arrays, waveforms, string variables, and/or string arrays depending on what the particular driver allows. Multiple targets are separated by commas. Compound targets, which are used by the high-level IEEE 488 Instrument driver, INS.SPS, contain semicolons. They allow the transfer of composite data forms such as ASCII and numeric instrument settings. (INS.SPS is not supported by TEK SPS BASIC VØ2-Ø1.)

The specification following the keyword **FROM** designates the source instrument. If a pound sign (#) is used, the expression after it is the instrument logical unit number (ILUN) of the ATTACHed instrument. The optional semicolon and expression is used by the high-level IEEE 488 Instrument driver, INS.SPS, to specify the secondary address of the source IEEE 488 instrument. (INS.SPS is not supported by TEK SPS BASIC VØ2-Ø1.) The optional string expressions are the driver-dependent strings which determine what data or status information is acquired.

If an at sign (@) is used, the expression following it is the number of the IEEE 488 interface through which the data or information is acquired. When the at sign is specified, the low-level IEEE 488 Interface driver, GPI.SPS is used. The expressions after the interface number are the primary talk address and the optional secondary address of a device connected to the IEEE 488 Interface Bus. See Section 6 for complete documentation.

GETBLK (Nonresident)

Examples:

```
150 GETBLK DK1:"TEST.DAT",3,B(Ø:255)
460 GETBLK X,A
320 GETBLK DX:J*2,A$
```

Syntax Form:

```
[line no.] GETBLK [device name[constant]:][string expression,]
                                 (string variable)
                     expression, {               }
                                 (array          )
```

Descriptive Form:

```
[line no.] GETBLK [device name[drive number]:][file name,]
                                   (target string variable)
                     block number, {                      }
                                   (target array          )
```

Purpose:

To obtain the contents of a block from a directory-structured peripheral device.

Discussion:

The GETBLK command obtains a block of data from a directory-structured device. (One block contains 256 16-bit words of data. One word can hold one 16-bit integer or two 8-bit ASCII characters.) The block obtained can be specified as an absolute block number of the device or as a block relative to the start of a file. Depending on the argument used, the data is returned in either a 256-element array or a 512-character string.

Using the Syntax Options:

The device must be directory-structured. If no device is named, the system device is used. If the named device does not use the system device driver, its driver must be LOADed before GETBLK executes. If the drive number is omitted, zero is assumed.

Which block of data is obtained is determined by the expression and the presence or absence of a file name. The expression, which must evaluate to a non-negative number, is rounded to an integer. If the file name is omitted, that integer is used as an absolute block number of the device and its contents are obtained. If a file name is given, that integer is added to the file's starting block number to produce the number of the block returned. In either case, the resulting block number must be in the range from zero to the largest block number of the device, inclusive.

The contents are returned in either an array or a string, depending on which is specified. If an array is used, it must be dimensioned or zoned to 256 elements to exactly hold the contents of the block. The array may be either floating-point or integer, but you can save memory space by specifying an integer array. If a string variable is used, the contents are returned as a 512-character string.

GETFREE (Nonresident)

Examples:

```
   GETFREE Y
15 GETFREE S,X
```

Syntax Form:

```
[line no.] GETFRE variable[,floating-point variable]
```

Descriptive Form:

```
[line no.] GETFREE target for amount of free memory
                   [,target for amount of free extended memory]
```

Purpose:

To obtain the amount of free memory available.

Discussion:

GETFREE obtains the number of words of controller memory that is free for program text, data storage, nonresident commands, and drivers. **The number of words available for array storage in extended memory (XM) systems also can be obtained using GETFREE.** Before GETFREE calculates the amount of free memory, it compresses the string storage area in order to get the most free memory available. Since this is a nonresident command, the number of words of memory needed for the GETFREE command reduces the total of free memory by that amount.

Using the Syntax Options:

The number of words of free memory is returned in the first argument, a variable. Specifying the optional second argument, a floating-point variable, also returns the number of free words of array storage available in extended memory systems. If the optional argument is used with standard memory systems, a zero is returned. (GETFREE VØ2-Ø1 does not support the optional second argument.)

Application Example:

Suppose you have a choice of two routines to do the same thing. One is fast, but requires a large amount of memory; the other is slow, but uses much less memory. Assuming you want to use the faster routine if possible, you put that in your program's overlay section. Then using the GETFREE command to determine the available free memory space, your program can OVERLAY the slower routine if necessary.

In the example below, N is the target value for the amount of free memory; W is the number of words the fast routine requires to execute. After finding N, the program RELEASEs the GETFREE command to reclaim that space. Then it compares N, plus S, the size of the GETFREE command, with W. Thus, only if the remaining memory is too small does it OVERLAY the fast routine with the slow routine. The fast routine in the overlay area (lines 1ØØØ to 1999) is DELETEd before the slow routine is brought in because the slow routine might not OVERLAY all the lines of the fast routine.

```
550 GETFREE N
560 RELEASE "GETFREE"
570 IF W<N+S THEN 610
580 REM DELETE FAST ROUTINE; OVERLAY THE SLOW, SMALL ROUTINE
590 DELETE 1000,1999
600 OVERLAY DX1:"SMALL"
610 REM CONTINUE
```

Normally if line 57Ø were true, the line it transfers control to would _not_ be a REM statement but an _executable_ statement beyond the section to be skipped. A REM statement was used here just to demonstrate a method.

GETLINE (Nonresident)

Examples:

```
50 GETLINE Y
60 GETLINE A(I,J+K)
```

Syntax Form:

[line no.] **GETLIN** variable

Descriptive Form:

[line no.] **GETLINE** target variable

Purpose:

To obtain the line number of the currently executing line.

Discussion:

GETLINE is used to determine where program control is when the command is executing. GETLINE statements are especially useful when you are debugging a program, since it allows you to follow the flow of program execution by printing the returned line numbers on the terminal or by storing them in an array.

If GETLINE is issued in the immediate mode, zero is returned.

Application Example:

To follow program flow from the terminal, you can concatenate two commands like:

```
GETLINE L\PRINT L
```

to each possible destination of a branch statement. Here the line number is returned in L. For example:

```
 1ØØ GOSUB N OF 1ØØØ,2ØØØ
 11Ø X=Y+Z\GETLINE L\PRINT L
  .
  .
  .
 49Ø IF X> 9ØØ THEN 4455
  .
  .
  .
1ØØØ X=SIN(T*W)\GETLINE L\PRINT L
  .
  .
  .
2ØØØ X=SIN(X)\GETLINE L\PRINT L
  .
  .
  .
4455 X=X/CF\GETLINE L\PRINT L
```

would print the line number of the statement jumped to every time there was a branch.

To follow the change in a variable along with the line number where the change occurred, use a similar technique. For example, to follow the changes in X concatenate something like:

```
GETLINE A(N)\B(N)=X\N=N+1
```

to the end of each line that changes X. Here, the line number and the X value are stored in arrays, A and B, that have been previously dimensioned to the same size.

After running the program, executing:

```
FOR I=Ø TO SIZ(A)-1\PRINT A(I),"X=";B(I)\NEXT I
```

would print the line numbers and their corresponding X values on the terminal.

GETLOC (Nonresident)

Examples:

```
180 GETLOC "177650",P
190 GETLOC X*2,M,0,6
```

Syntax Form:

$$\text{[line no.] } \textbf{GETLOC} \left\{\begin{array}{l}\text{expression}\\ \text{string expression}\end{array}\right\} \text{,floating-point variable[,expression,expression]}$$

Descriptive Form:

$$\text{[line no.] } \textbf{GETLOC} \left\{\begin{array}{l}\text{decimal address}\\ \text{octal address}\end{array}\right\} \text{,target variable for contents of address}$$

[,low-order bit number for obtaining segment of contents, high-order bit number]

Purpose:

To allow examination of the contents of any controller memory location or valid device or register address.

Discussion:

The GETLOC command is used by the PATCH files. This command is not intended for general use.

The GETLOC command returns the contents of a controller memory location or a valid interface address (explained below). The contents of the specified address are returned in a floating-point variable. An even address references a word (16 bits); an odd address, a byte (8 bits). Optionally, a segment of the requested word or byte address can be examined.

Valid Addresses:

Standard Memory Systems. One word (16 bits) can produce 2^{16} unique addresses -- 0 to 177777 octal. With byte addressing, this means you can reference one of 64K distinct bytes (32K words) with a 16-bit address. However, the highest 4K possible word addresses are reserved as peripheral address space, leaving a maximum of 28K word addresses to use for controller memory locations. Thus, for GETLOC, the valid addresses are the controller memory locations (0 to 157777 octal) plus those addresses in the peripheral address space to which interfaces are strapped.

Extended Memory Systems. Systems with memory-management hardware and TEK SPS BASIC V02XM software have 18-bit addresses. This permits 256K byte (128K word) addresses of which the highest 4K word addresses are reserved as peripheral address space. (This means, for example, that a peripheral status register which is addressed as 164100_8 in a standard memory system must be addressed as 764100_8 in an XM system.) For extended memory (XM) systems, the valid addresses include the possible controller memory locations (0 to 757777 octal) plus any addresses in the peripheral address space to which interfaces are strapped.

If the specified address is not valid for your system or controller, a fatal error is issued.

Using the Syntax Options:

The first argument is the examined address. If the argument is a string, it must be the desired octal address. A string expression should evaluate to a string of no more than eight octal digits. However, in standard memory systems only the lower 16 binary digits (bits) of the value represented by the string are used as the address. In extended memory (XM) systems, the lower 18 bits are used. If the argument is numeric, it must be the decimal equivalent to the desired address. A numeric expression is converted to binary and, if necessary, truncated to a 24-bit binary integer. Again, in standard memory systems only the lower 16 bits are used as the address; while in extended memory systems, the lower 18 bits are used. In any case, if the address is even, an entire 16-bit word is referenced, if the address is odd, only an 8-bit byte.

The second argument must be a floating-point variable. After execution of the command, it contains the contents of the specified address as a floating-point number.

The optional third and fourth arguments are used to examine the contents of a segment of the specified address. The third argument designates the low-order bit number of the desired memory word or byte; the fourth, the high-order bit number. The bits are numbered from zero. The range of the segment must be Ø to 15 for an even (word) address and Ø to 7 for an odd (byte) address.

GETPRIORITY (Nonresident)

Examples:

```
150 GETPRI K
130 GETPRI K(VAL(A$))
```

Syntax Form:

[line no.] **GETPRI** variable

Descriptive Form:

[line no.] **GETPRI**ORITY target variable

Purpose:

To obtain the priority of the task being executed.

Discussion:

Statements are executed on the basis of their assigned or default priority value. (The Scheduler's priority-based execution process is discussed in Section 1.) Assigned priorities range from Ø (lowest) to 126 (highest). BASIC runs at a default priority of 5Ø, but the PRIORITY command can be used to alter the system priority. Also, external interrupt routines and routines scheduled by the SCHEDULE, RESCHEDULE, and WHEN commands can have different priorities, either by specification or default.

When the GETPRIORITY command executes, it returns the system priority in the specified variable. (Since the system priority is the priority of the currently executing statement, a GETPRIORITY statement returns the priority at which it executes.) This information can be used when setting interrupt routine priorities.

Application Example:

This command can be used to insure that an interrupt routine executes at a higher priority than the current system priority. First, execute the GETPRI command, then set the interrupt priority one higher than the returned value. For example:

```
100 GETPRI N
110 WHEN #1 HAS A$ AT N+1 GOSUB 1000
```

GOSUB

Examples:

```
200 GOSUB 1000
500 GOSUB X*2 OF 1000,2200,1200,1500
```

Syntax Form:

$$\text{[line no.] } \textbf{GOSUB} \begin{Bmatrix} \text{line number} \\ \text{expression } \textbf{OF} \text{ line number[,line number] ...} \end{Bmatrix}$$

Descriptive Form:

$$\text{[line no.] } \textbf{GOSUB} \begin{Bmatrix} \text{line number} \\ \text{line number selector } \textbf{OF} \text{ line number list} \end{Bmatrix}$$

Purpose:

To transfer program control either unconditionally to a single subroutine or to one of a list of subroutines.

Discussion:

A subroutine is a sequence of statements terminated by a RETURN statement. Subroutines are useful when certain program actions must be repeated several times in a program. The program segment can be written once and then activated (called) by any part of the main program any number of times. A subroutine may call another subroutine, or even call itself. The RETURN statement signals the end of the subroutine. Several RETURN statements may appear in a subroutine.

The GOSUB command calls (transfers control to) the subroutine. When the subroutine is finished executing (a RETURN command is encountered), program control returns to the next command following the GOSUB. (If a statement follows the GOSUB on the same line as the GOSUB, control returns to it, not to the next line of code.)

GOSUB calls a subroutine by transferring program control to a line number which is assumed to be the first line of the subroutine. The transfer can be either an unconditional GOSUB to a single, specified line number or a computed GOSUB to one of a list of line numbers. When the computed GOSUB is used, the current value of the expression is computed and program control transfers to the line whose position in the list corresponds to that value. For example, if the expression evaluates to 2, control passes to the second line number in the list. So in example line 5ØØ above, if the expression X*2 equals 2, control goes to the subroutine beginning at line 22ØØ. When the current value of the expression is greater than the number of line numbers in the list or less than 1, the GOSUB is ignored and control passes to the statement following the GOSUB command. No warning error is issued.

The line number specified or selected by the value of the expression must be the line number of a statement in memory when the GOSUB command executes.

Using the Syntax Options:

When only a line number is specified, program control is unconditionally transferred to the subroutine indicated by that line number.

When an expression and the keyword **OF** are specified, the transfer of program control is to one of a list of line numbers (subroutines). The value of the expression, when evaluated and rounded to an integer, determines which subroutine is called. If the value is in the range from 1 to n where n is the number of line numbers in the list, control transfers to one of these line numbers -- the line number whose position corresponds to that value. However, if the value is out of range (i.e., less than 1 or greater than n) control passes to the statement following the GOSUB.

GOTO

Examples:

```
500 GOTO 675
100 GOTO X/2 OF 500,450,780
```

Syntax Form:

```
                  (line number                                     )
[line no.] GOTO  <                                                  >
                  (expression OF line number[,line number] ...     )
```

Descriptive Form:

```
                  (line number                                )
[line no.] GOTO  <                                             >
                  (line number selector OF line number list   )
```

Purpose:

To transfer program control either unconditionally to a single, specified line number or to one of a list of line numbers.

Discussion:

Normally, BASIC statements execute in the order of their line numbers starting with the lowest line number in memory. The GOTO statement overrides the normal flow of program execution, transferring program control to a line other than the next sequential line of text. The transfer can be either an unconditional GOTO to a single, specified line number or a computed GOTO to one of a list of line numbers. When the computed GOTO is used, the current value of the expression is computed and program control transfers to the line whose position in the list corresponds to that value. For example, if the expression evaluates to 3, control passes to the third line number in the list. In example line 100 above, if the expression X/2 equals 3, control goes to line 780. When the current value of the expression is greater than the number of line numbers in the list or less than 1, the GOTO is ignored and control passes to the statement following the GOTO command. No warning error is issued.

The line number specified or selected by the value of the expression must be the number of a statement in memory when the GOTO command executes.

Using the Syntax Options:

When only a line number is specified, program control is unconditionally transferred to that line number.

When an expression and the keyword **OF** are specified, program control transfers to one of a list of line numbers. The value of the expression, when evaluated and rounded to an integer, determines where program control goes. If the value is in the range from 1 to n where n is the number of line numbers in the list, control transfers to one of these line numbers -- the line number whose position corresponds to that value. However, if the value is out of range (i.e., less than 1 or greater than n) control passes to the statement following the GOTO.

Application Example:

A computed GOTO can be used in a dispatch routine that sends control to one of several places in the program depending on the current value of an expression. For example:

```
100 PRINT "ENTER YOUR CHOICE, 1 TO 5"
110 INPUT N$
120 GOTO ASC(N$)-ASC("0") OF 500,550,700,400,200
130 PRINT "NUMBER OUT OF RANGE, TRY AGAIN"
140 GOTO 100
 .
 .
 .
```

Here the transfer depends on what character is entered from the keyboard. By allowing the number to be INPUT as a string (line 110), a nonnumeric response will not cause an error. Then, by using the ASCII function (ASC) in the expression in line 120, only a correct choice lets the program continue beyond line 140. When a string beginning with a 1,2,3,4, or 5 is entered, transfer is made to one of the five line numbers in the list (line 120). With any other character the value of the expression is out of range and control passes to line 130. There, a message is printed and then the unconditional GOTO (line 140) sends control back to line 110 where another response is solicited.

HASH (Nonresident)

Examples:

```
950 HASH A$(K),LL,P(K)
770 HASH "Ø2920",197,J
```

Syntax Form:

```
[line no.] HASH string expression,expression,variable
```

Descriptive Form:

```
[line no.] HASH key, table size, target variable for index
```

Purpose:

To convert a string (hash key) to an index number that can be used to access an indexed list such as an array or a record I/O file.

Discussion:

The HASH command can be used when inserting or retrieving data stored in an indexed list such as an array or a record I/O file. It provides a tool that allows faster access to the data stored in the list than can be achieved by a simple search of the list.

The concept of hashing is this: Each unit of data (i.e., each array element or each file record) is assigned its own key, such as an I.D. number. This key is converted by a hashing function (in this case, the HASH command) to an index that points to a position in the list. Then the index is used to store the data unit. To retrieve the data, the same hashing function (i.e., the same HASH statement) is used to convert the key to the index again.

HASH always returns the same index for a given key and indexed-list length. However, HASH may map more than one key to the same index number. For this reason, when adding an item to the list, check the contents of the position pointed to by the index to be sure that it is empty. Likewise, when retrieving data, check that the data unit pointed to is the one sought.

If, when adding to an indexed list, the index returned by HASH points to a list position that is already in use, the data should be stored using another method. One of the simplest is to put the data in the next empty position. (See the example routines.) But, whatever is done in this case, the same method must be used to retrieve the data when the index does not point to the desired item.

The HASH command uses an algorithm equivalent to this BASIC routine where H$ is the hash-key string, N is the number of data units (elements or records) in the indexed list, and I is the index.

```
110 I=0
120 FOR P=1 TO LEN(H$)
130 I=I+ASC(SEG(H$,P,P))*2^(P-1)
140 NEXT P
150 IF I>=N THEN I=I-ITP(I/N)*N
160 RETURN
```

The index number returned is modulo N (i.e., $0 \leq I \leq N-1$). Notice that the longer the hash key is, the longer the execution time will be.

This algorithm results in a reasonably even distribution of index numbers. The distribution is best when the number of list items (possible indexes) is a prime number. (For more information on hashing and hashing functions see a book such as D.E. Knuth's The Art of Computer Programming Volume III.)

NOTE

The HASH command is not available with TEK SPS BASIC VØ2-Ø1.

Using the Command Syntax:

The string expression specifies the hash key. Each data unit (array element or file record) should have its own unique key.

The expression is the number of data units in the indexed list (the number of elements in the array or the number of records in the file). When evaluated and rounded to an integer, it must be greater than zero.

The index that the hash key maps to is returned in the variable.

Application Example:

Here are two routines that use HASH to determine which record I/O file data record to write or to read. Since each data record consists of a 67 character string (2Ø for a name, 4Ø for an address, and 7 for a phone number) the data file is created by a statement like:

```
DEFINE DX:"ADDR.DAT" AS STG 67 WITH 53
```

In these routines, the hash key for each data unit (record) is the name stored with the other information. The table size used in the HASH statement is the number of records in the file, 53.

```
1ØØ REM STORE A RECORD USING HASH
11Ø REM
12Ø REM OPEN THE FILE
13Ø OPEN #1 AS DX1:"ADDR.DAT" FOR UPDATE
14Ø REM GET NAME
15Ø PRINT "ENTER: FIRST NAME, LAST NAME"
16Ø INPUT NA$
17Ø REM USE NAME AS HASH KEY
18Ø HASH NA$,53,I
19Ø REM CHECK FOR EMPTY RECORD
2ØØ READU #1<I>,S$=67
21Ø IF ASC(SEG(S$,1,1))=Ø THEN GOTO 36Ø
22Ø REM SEARCH FORWARD TO END OF FILE FOR EMPTY RECORD
23Ø FOR N=I+1 TO 52
24Ø READU #1<N>,S$=67
25Ø IF ASC(SEG(S$,1,1))=Ø THEN GOTO 34Ø
26Ø NEXT N
```

```
270 REM SEARCH BACKWARD TO BEGINNING OF FILE FOR EMPTY RECORD
280 FOR N=I-1 TO Ø STEP -1
290 READU #1<N>,S$=67
300 IF ASC(SEG(S$,1,1))=Ø THEN GOTO 340
310 NEXT N
320 PRINT "FILE FULL"
330 GOTO 440
340 I=N
350 REM GET REST OF INFORMATION
360 PRINT "ENTER: STREET ADDRESS"
370 INPUT A1$
380 PRINT "ENTER: CITY, STATE, ZIP"
390 INPUT A2$
400 PRINT "ENTER: PHONE NUMBER
410 INPUT PH$
420 REM WRITE RECORD IN FILE
430 WRITEU #1<I>,NA$=2Ø,A1$=2Ø,A2$=2Ø,PH$=7
440 CLOSE #1
450 RETURN
500 REM
510 REM READ A RECORD USING HASH
520 REM
530 REM OPEN THE FILE
540 OPEN #1 AS DX1:"ADDR.DAT" FOR UPDATE
550 REM GET NAME
560 PRINT "ENTER: NAME EXACTLY AS STORED"
570 INPUT NA$
580 REM USE NAME AS HASH KEY
590 HASH NA$,53,I
600 REM CHECK FOR A MATCH
610 READU #1<I>,S$=67
620 IF TRM(SEG(S$,1,2Ø))=NA$ THEN GOTO 770
630 REM IF NAME DOESN'T MATCH:
640 REM SEARCH RECORDS TO END OF FILE FOR A MATCH
650 FOR N=I+1 TO 52
660 READU #1<N>,S$=67
670 IF TRM(SEG(S$,1,2Ø))=NA$ THEN GOTO 770
680 NEXT N
690 REM SEARCH BACKWARD TO BEGINNING OF FILE
700 FOR N=I-1 TO Ø STEP -1
710 READU #1<N>,S$=67
720 IF TRM(SEG(S$,1,2Ø))=NA$ THEN GOTO 770
730 NEXT N
```

```
74Ø PRINT NA$;" NOT IN FILE"
75Ø GOTO 81Ø
76Ø REM PRINT CONTENTS OF RECORD
77Ø PRINT NA$
78Ø PRINT TRM(SEG(S$,21,4Ø))
79Ø PRINT TRM(SEG(S$,41,6Ø))
8ØØ PRINT SEG(S$,61,67)
81Ø CLOSE #1
82Ø RETURN
```

The first routine (lines 1ØØ to 45Ø) stores the data records. If the index (record number) returned by HASH points to an empty record, the data is written in that record. If the index points to a nonempty record, the routine searches for the next empty record and writes the data there. The search is made from the individual record to the end of the file. If no empty record is found in this first search, a second search is made from the indicated record to the beginning of the file. Obviously if no empty record is found on the second search, the file is full.

As you can see, as the file begins to fill, the speed advantage of hashing over a sequential search is lost. For that reason, this technique is seldom used with densely filled files. Instead, HASH is used to access a hash table that is filled with pointers to where the data is stored. Data records with hash keys that map to the same hash-table index are linked together. However, such an example is beyond the scope of this manual.

The second routine (lines 5ØØ to 82Ø) retrieves the information from the file. The same kind of logic is used to find the desired record as was used to store the record, except this time a particular name is sought instead of an empty record. Thus, if the record pointed to does not contain the desired name, a serial search is made first forward and then backward through the file. But, the given name must be <u>exactly</u> as it is stored or no match will be found.

HOOK (Nonresident)

Examples:

```
    HOOK
100 HOOK DX1:
    HOOK DK: FOR RT11 "MONITR.SYS"
```

Syntax Form:

[line no.] **HOOK** [device name[constant]**:**] [**FOR RT11** string expression]

Descriptive Form:

[line no.] **HOOK** [name of bootable device[drive number]**:**] [**FOR RT11** file name]

Purpose:

To install either the SPS or the DEC RT-11 bootstrap program in the bootstrap blocks of the specified disk.

Discussion:

The SPS bootstrap program is an absolute loader for a file containing Resident BASIC, the software initialization routines, and the driver for the HOOKed device. This file has a name with the form SPSxx.LDA, where xx is the system device after that file is loaded (e.g., SPSDK.LDA is the BASIC monitor file with DK as the system device). Such files are called SPS .LDA files.

The HOOK command (without the FOR RT11 option) installs (writes) the appropriate SPS bootstrap program into the bootstrap block of the disk in the specified device. For example:

```
HOOK DX1:
```

installs the absolute loader for SPSDX.LDA on the floppy disk in drive 1.

After a peripheral device has been HOOKed, the BOOT command or the ROM bootstrap loader in the controller may be used to bootstrap that device. However, installing the SPS bootstrap program on a peripheral device does not automatically assure that TEK SPS BASIC will be loaded the next time that device is booted as the system device. The appropriate file (e.g., SPSDK.LDA, SPSDX.LDA, SPSDL.LDA, or SPSDY.LDA) must be present on the disk before BASIC can be loaded.

[The DEC RT-11 bootstrap program loads a file with the ".SYS" file name extension that contains the DEC RT-11 Monitor. This file must be on the named device before the DEC RT-11 bootstrap program can be installed. The HOOK command with the FOR RT11 option writes the first blocks of the .SYS file into the bootstrap blocks of the disk, installing the DEC RT-11 bootstrap program.]

The HOOKQ command installs a general bootstrap program that loads any .LDA file on the disk. See the HOOKQ command for more discussion.

Using the Syntax Options:

The device being HOOKed must be one of the bootable devices such as DX, DK, DL, or DY. When no device is named, the system device is used. Unless the device is DX or DK, the peripheral overlay file for that device (e.g., DL.OVL or DY.OVL) must be on either the system device or the disk being HOOKed. If the device being HOOKed does not use the system device driver, its driver must be LOADed into memory before the HOOK command executes. If the device number is omitted, zero is assumed.

NOTE

The DL and DY drivers are not supported by TEK SPS BASIC VØ2-Ø1.

[When the optional keywords **FOR RT11** are used, the DEC RT-11 absolute loader is installed. This feature allows you to recover the DEC RT-11 Monitor on a disk after you have HOOKed it for TEK SPS BASIC. The appropriate file name with the .SYS extension must follow the keywords FOR RT11. The named file should be the DEC RT-11 Monitor file. Its name depends on the version of software. For versions 1 and 2 of the DEC RT-11 Monitor, the file is "MONITR.SYS"; for version 3, the file is "RKMNSJ.SYS". Omitting this option installs the proper SPS absolute loader on the disk.]

HOOKQ (Nonresident)

Examples:

```
HOOKQ
HOOKQ DX1:
HOOKQ DK4:
```

Syntax Form:

```
[line no.] HOOKQ [device name[constant]:]
```

Descriptive Form:

```
[line no.] HOOKQ [name of bootable device[drive number]:]
```

Purpose:

To install, on a disk, a bootstrap program which loads files that have the .LDA extension.

Discussion:

The bootstrap program (absolute loader) is installed in the bootstrap blocks of the designated disk. After a bootstrap is installed by HOOKQ, the BOOT command or the ROM bootstrap loader in the controller can be used to bootstrap the device. When the device is booted, a prompt (*) is printed on the terminal. Any file on the specified device with an .LDA extension can then be loaded by entering its file name followed by a carriage return. When entering the file name, **do not include the .LDA extension.**

The bootstrap program does not provide Rubout or Control-U capabilities. If you make a mistake while typing the file name, enter a carriage return. The message "FILE NOT FOUND" will be printed followed by another prompt for the file name.

Using the Syntax Option:

The specified device must be one of the bootable devices such as DX, DK, DL, or DY. When no device is named, the system device is assumed. Unless the device is DX or DK, the peripheral overlay file for that device (e.g., DL.OVL or DY.OVL) must be on either the system device or the disk being HOOKQed. If the specified device is not the system device, its driver must be LOADed into memory before the HOOKQ command executes. When the drive number is omitted, the disk in drive Ø is HOOKQed.

NOTE

The DL and DY drivers are not supported by TEK SPS BASIC VØ2-Ø1.

IF

Examples:

```
1ØØ IF A=B THEN GOTO 5ØØ
12Ø IF A$ = "STOP" THEN STOP
13Ø IF Q*5<=P/L THEN 45Ø
```

Syntax Form:

$$\text{[line no.] } \textbf{IF} \left\{ \begin{array}{l} \text{expression} \\ \text{string expression} \end{array} \right\} \text{ relational operator } \left\{ \begin{array}{l} \text{expression} \\ \text{string expression} \end{array} \right\}$$

$$\textbf{THEN} \left\{ \begin{array}{l} \text{statement} \\ \text{line number} \end{array} \right\}$$

Purpose:

To conditionally transfer program control to a specified line, or conditionally execute a statement.

Discussion:

When an IF statement is executed, the two expressions (or two string expressions) are evaluated and then compared with each other. Depending on the relational operator used, the result is either true or false. If the result is true, the statement following the keyword **THEN** is executed or if only a line number is specified, a GOTO to that line number is performed. If the result is false, the next sequential statement is executed. The next statement may either be concatenated to the same line of text as the IF statement or be the next line of text.

The relationship operators used to determine if the condition is true or false are:

Operator	**Meaning**
=	equal
<	less than
<= or =<	less than or equal
>	greater than
>= or =>	greater than or equal
>< or <>	not equal

Strings are compared, character for character, from left to right. The first inequality determines the result. The string with the higher ASCII value at that character is considered the larger. Example: SUNRISE and SUNNY; since R has a larger ASCII value than N, SUNRISE is larger than SUNNY. If the two strings are of unequal length, and the end of the shorter string is found before an inequality, the shorter string is assumed to be the smaller (less). Example: RUN and RUNNING; RUN is less than RUNNING. (Appendix A contains the complete ASCII character set and the corresponding decimal values of each character.)

Using the Syntax Options:

The two expressions must be of the same type: either both numeric expressions or both string expressions. Only single variables may be compared. Thus, array elements can be specified, but not arrays.

If a command is specified after the keyword **THEN**, it can be any resident or nonresident command -- including another IF command. When a line number is specified, it must be the number of a statement in memory when the IF command executes.

Application Example:

Since any statement can follow the THEN in an IF statement, several IF statements can be strung together to perform a logical AND operation. That is, if the first expression is true, and if the second expression is true, ... and if the nth expression is true, then and only then perform the statement following the final THEN. Otherwise, go to the next sequential statement.

Consider the following IF statement:

```
1ØØ IF Y=X THEN IF Y=Z THEN 55Ø
```

This statement transfers program control to line 55Ø if the variable Y is equal to both the variables X _and_ Z.

To perform a logical OR operation, separate IF statements can f[illegible]w one another sequentially. For example:

```
1ØØ IF Y=X THEN 55Ø
11Ø IF Y=Z THEN 55Ø
```

These statements transfer program control to line 55Ø if Y equals either X or Z.

IGNORE (Nonresident)

Examples:

```
150 IGNORE #N,TASK 2
170 IGNORE #3,A$
900 IGNORE #5
340 IGNORE @1,"SRQ"
570 IGNORE TASK 2
280 IGNORE ALL
```

Syntax Form:

```
                      ( { # expression } [ , { TASK expression      } ] )
                      ( { @ expression } [   { string expression    } ] )
[line no.] IGNORE     (   TASK expression                               )
                      (   ALL                                           )
```

Descriptive Form:

```
                      ( { # ilun                        } [ , { TASK task number           } ] )
                      ( { @ IEEE 488 interface number   } [   { driver-dependent interrupt } ] )
[line no.] IGNORE     (                                   [   {   condition specification  } ] )
                      (   TASK task number                                                     )
                      (   ALL interrupt conditions for all iluns                               )
```

Purpose:

To cancel the action of WHEN commands so all or specific subsequent interrupts are ignored.

Discussion:

The WHEN command sets up a structure that allows a program to recognize an instrument interrupt and to schedule user-written interrupt subroutines when the interrupt occurs. The IGNORE command cancels (nullifies the action of) one or more WHEN statements with the result that one or more interrupts are ignored.

The IGNORE command has no effect on interrupts that have already occurred and have already caused the interrupt routine to be scheduled for execution. To ignore all interrupts and clear the Scheduler, use a STOP, END, or Control-P. (The function and parts of the Scheduler are explained in Section 1.)

Using the Syntax Options:

The instrument logical unit number (ILUN) following the pound sign (**#**) represents a particular instrument. The expression following the at sign (**@**) represents an IEEE 488 interface number. This second convention is used when communication is with the low-level IEEE 488 Interface driver ("GPI.SPS") discussed in Section 6. The ILUN or interface number indicates from which instruments or interface to IGNORE an interrupt. If no specification follows either of these, all WHENs specifying that ILUN or interface number are canceled so all subsequent interrupts from that instrument or interface are ignored.

The string expression is a driver-dependent interrupt specification. It must be a string accepted by the driver for the specified ILUN or interface. When used, the interrupt from the ILUN or interface as specified by the string is IGNOREd. Other interrupts from that ILUN or interface are not affected.

The expression following the keyword **TASK** is the task number of the interrupt subroutine. This number, when rounded to an integer, must be between 0 and 126, inclusive. If the TASK option is used with either the ILUN or interface specification, all WHEN conditions associated with that task number and ILUN or interface number are canceled. If the TASK option is used alone, all WHEN conditions associated with the task number are canceled.

When the keyword **ALL** is used, all WHEN conditions are canceled.

The options and resulting actions of the IGNORE command are summarized in the table.

Summary of IGNORE Command

Specification	**Which WHEN is Ignored**
#ILUN	all WHENs with that ILUN
#ILUN, interrupt specification string	the WHEN with that ILUN and that string
#ILUN, **TASK** number	all WHENs with that ILUN and that task number
@IEEE 488 interface number	all WHENs with that interface number
@IEEE 488 interface number, interrupt specification string	the WHEN with that interface number and that string
@IEEE 488 interface number, **TASK** number	all WHENs with that interface number and that task number
TASK number	all WHENs with that task number
ALL	all WHENs

INPREQ (Nonresident)

Examples:

```
150 INPREQ GOSUB 7000
160 INPREQ CHAR, NOECHO GOSUB 670
170 INPREQ CHAR GOSUB 440
180 INPREQ
```

Syntax Form:

```
                      [[CHAR        ]                    ]
[line no.] INPREQ     [[NOECHO      ]  GOSUB line number ]
                      [[CHAR,NOECHO ]                    ]
```

Descriptive Form:

```
                      [[CHARacter          ]                    ]
[line no.] INPREQ     [[NOECHO             ]  GOSUB line number ]
                      [[CHARacter, NOECHO  ]                    ]
```

Purpose:

To permit unsolicited input from the keyboard during program execution. When the input occurs, program control transfers to a subroutine to INPUT it.

Discussion:

TEK SPS BASIC offers two ways to enter data from the keyboard to an executing program via an INPUT statement -- with or without an input request enabled. The normal (default) condition is that an input request is not enabled. An input request is enabled by an INPREQ statement.

If an input request is not enabled (the default condition), the INPUT command is used to solicit program data from the keyboard for a running program. When an INPUT statement is encountered, a question mark (?) is printed on the terminal prompting the user to enter data from the keyboard. Until then, if the keyboard is live (not locked by the LOCKKB command), any input from the keyboard is interpreted to be new program text or an immediate mode command.

If an input request is enabled by the INPREQ command, you can enter unsolicited program data before an INPUT statement is executed. However, then you cannot enter a line of program text or an immediate mode command while a program is running. When an input request is enabled, all input from the keyboard during program execution is interpreted to be program data. When the data is entered from the keyboard, program control transfers to a subroutine. The assumption here is that the subroutine which receives control has an INPUT statement that uses the data, assigning the data to its variables. That INPUT statement must be set up to expect data from the keyboard. (The INPUT statement's peripheral logical unit number must be omitted or be zero for the keyboard.)

The unsolicited input can be either a single character or a full line of data. When a line of data is required, it must be terminated by a carriage return. If a single character causes the transfer to the subroutine, no carriage return is needed.

The data in a line of unsolicited input must follow the same rules as solicited input. A line may contain several numeric values separated by commas; but, because strings are terminated by a carriage return, only one string value is permitted per line. Also, the data types must match the variables in the INPUT statement. See the INPUT command for a complete discussion of data types and variables.

If the INPUT statement needs more data than is supplied by the line (or character), the program waits for additional input from the keyboard. However, no question mark (?) is printed to tell you that this is the case. In fact, while an input request is enabled, no INPUT statement in the program will print a question mark to prompt data input. If a line of unsolicited input supplies more data than is needed by the first INPUT statement in the subroutine, the extra data is ignored.

You can also permit or suppress echoing of the character(s) as they are entered for program input. In other words, you can choose whether or not to have the data printed on the terminal as it is typed in.

If an INPUT statement is encountered in the program before the input request is satisfied, the program waits until the input is completed. Then that INPUT statement gets the data, and program control does not pass to the subroutine. If more data is entered in a line of input than the INPUT statement wants, the extra data is discarded.

An input request remains enabled -- even after the data has been assigned by an INPUT statement -- until cancelled. Executing an INPREQ statement that has no arguments cancels the input request. Any unassigned data that had been accepted as unsolicited input is discarded. CHAIN, OLD, STOP, END, DELETE TEXT, DELETE ALL, or Control-P also cancels any input requests and discards any unassigned data that may have been entered.

[INPREQ transfers control to the subroutine by scheduling the specified line number with the same task number as the current task and at a priority one greater than the current task. However, if the current task's priority is 126, the subroutine is scheduled with that same priority (126). In that case, the subroutine does not gain control until after the current task terminates or the priority of the current task is lowered below 126 by a PRIORITY statement.]

Using the Syntax Options:

An **INPREQ** statement with no arguments cancels any previous input request.

The optional keyword **CHAR** permits the entering of a single character (with no carriage return) to cause a transfer to the subroutine. If this keyword is omitted, entering a line of data (up to 79 characters terminated by a carriage return) causes the transfer to the subroutine.

The optional keyword **NOECHO** suppresses the echoing of the input. If NOECHO is specified, characters are not printed on the terminal when they are entered. If this keyword is omitted, each character is printed on the terminal as it is entered.

The line number following the keyword **GOSUB** is the starting line number of the subroutine which is transferred to after the INPREQ statement is satisfied.

INPUT (Nonresident)

Examples:

```
17Ø INPUT JJ,BV$
18Ø INPUT #5+C,OP
```

Syntax Form:

```
                                  ( variable         )  [   ( variable         ) ]
[line no.] INPUT [#expression,]  {  array            }  [ , {  array            } ] ...
                                 {  waveform         }  [   {  waveform         } ]
                                  ( string variable  )  [   ( string variable  ) ]
```

Descriptive Form:

```
                                  ( target variable         )  [   ( target variable         ) ]
[line no.] INPUT [# source plun,] {  target array            }  [ , {  target array            } ] ...
                                  {  target waveform         }  [   {  target waveform         } ]
                                  ( target string variable  )  [   ( target string variable  ) ]
```

Purpose:

To allow values to be assigned to variables. The values are ASCII characters, obtained from the terminal or other peripheral device or file.

Discussion:

The INPUT statement is used to enter data to a program while that program is running. The entry can be from the terminal (the default device) or from any other peripheral that can be OPENed FOR READ.

INPUT reads data that is in ASCII format rather than binary. Data entered from the keyboard and data stored on a peripheral device by the PRINT command are in this format. If a peripheral device other than the keyboard is used to enter data with the INPUT statement, be sure that the data is in ASCII format. (You can make a quick check of a file by COPYing that file to the terminal. If the file is in binary format, the output will be erratic.)

The data entered must match the type of variable specified in the INPUT statement. If a numeric variable is specified, a numeric string must be entered. This ASCII string is converted to a number and assigned to the numeric variable. If you enter an illegal character, such as a letter other than an E (which is used to signify a power of ten), an error results. With strings, you can enter any ASCII character except Control-P, Control-U, and carriage return. The maximum length of a string that can be INPUT from a file is 388 characters. The maximum length INPUT from the keyboard is 79 characters. (Longer strings can be created by concatenation.)

The INPUT command accesses a file or device by its peripheral logical unit number (PLUN), not by name. However, before a file or device other than the keyboard (PLUN zero) can be INPUT from, it must be OPENed FOR READ in order to assign a PLUN to that file or device. If the peripheral referenced by INPUT has been OPENed FOR WRITE or UPDATE, a fatal error results.

INPUT from the Keyboard:

If the INPUT is from the keyboard (and no INPREQ statement has been executed), a question mark (**?**) is printed on the terminal as a prompt to enter a piece of data. After the data is typed, followed by a carriage return, the data is assigned to the specified variable(s) in the order of their appearance in the INPUT statement. If more data is required than has been entered, another prompt (?) is printed. This continues until values for all the variables in the INPUT statement are obtained. (The question mark appears only when the terminal is used as the input device.)

When more than one numeric variable is specified in the INPUT statement, the **individual numbers can be separated by either commas or carriage returns.** For example, when the statement:

```
150 INPUT A,B,C
```

executes and you want to enter the numbers 1, 2, and 3, to be assigned to the variable A, B, and C, respectively. You may type either:

```
?1,2,3
```

or

```
?1
?2
?3
```

Remember, the question marks are printed on the terminal by the INPUT command as a prompt. (The entries are shown in bold.)

Strings (strings of ASCII characters, including letters, numbers, and punctuation) are simply typed in. No quotation marks are necessary (unless you want the quotation marks included in the string). If a string is the only argument specified in the INPUT statement, enter the string when the question mark is printed, and terminate the entry with a carriage return. If more than one string is specified, the input of **each string must be ended with a carriage return.**

String and numeric variables can be mixed in an INPUT statement. But remember, strings must be terminated with a carriage return. The following example illustrates how numbers and strings can be mixed. The items in bold are typed in from the keyboard.

```
LIST 15Ø\RUN
15Ø INPUT A,B,A$,B$,C
?3.45
?4.65,VOLTS
?SECONDS
?-1Ø.45E-Ø6
```

When line 15Ø executes, BASIC prints the ? on the terminal prompting you to enter data. After you enter the data, variables A and B are assigned the values 3.45 and 4.65, respectively. String variables A$ and B$ are assigned VOLTS and SECONDS, respectively. Variable C equals -1Ø.45E-Ø6.

CAUTION

When not using the INPREQ command, be sure to wait for the prompt (?) before entering data for an INPUT statement. Failure to do so may result in incorrect data being INPUT or even in deletion of program lines if the premature data is interpreted as a line number for system input. You will be able to know if this is the case because the non-idle mode system input prompt, an asterisk within parentheses **(*)**, will be displayed preceding the echo of the data as the data is typed in.

INPUT from a File:

Some of these same rules apply if the data entered is coming from a device other than the keyboard. INPUT expects data from a file or device to be as follows:

Values to be assigned to **numeric variables** must be terminated (delimited) by a **comma or a carriage return.**

Values to be assigned to **string variables** must be terminated (delimited) by a **carriage return.**

The discussion on the PRINT command explains how to PRINT data to a file so that it can be read by INPUT.

INPUT reads a file or device in a sequential manner starting at the beginning of the file with the first INPUT statement. Subsequent INPUTs from the same file continue reading data from where the previous INPUT ended.

Using the Syntax Options:

The expression following the pound sign (#) is the peripheral logical unit number (PLUN) from which the data is INPUT. The expression, when evaluated and rounded to an integer, must be between Ø and n, inclusive, where n is the number of PLUNs allowed at initialization time (default of six). When the pound sign and expression are omitted, the keyboard (PLUN zero) is assumed.

The list of variables to be assigned values by the INPUT statement may include integer or floating-point variables, integer or floating-point arrays, integer or floating-point waveforms, and string variables. **String arrays are not allowed.**

INTEGER

Example:

```
123 INTEGER A(5ØØ),TB(9),X(1ØØ,4)
```

Syntax Form:

$$\text{[line no.] } \textbf{INTEGER} \left\{\begin{array}{l}\text{simple numeric variable}\\ \text{integer array}\end{array}\right\} \text{(expression[,expression])}$$

$$\left[,\left\{\begin{array}{l}\text{simple numeric variable}\\ \text{integer array}\end{array}\right\} \text{(expression[,expression])}\right] \ldots$$

Descriptive Form:

$$\text{[line no.] } \textbf{INTEGER} \left\{\begin{array}{l}\text{simple numeric variable}\\ \text{integer array}\end{array}\right\} \text{(first dimension[,second dimension])}$$

$$\left[,\left\{\begin{array}{l}\text{simple numeric variable}\\ \text{integer array}\end{array}\right\} \text{(first dimension[,second dimension])}\right] \ldots$$

Purpose:

To allocate integer-format storage for arrays.

Discussion:

The INTEGER command functions like the DIM command except that each element of the array is defined as a one-word integer instead of a two-word, floating-point number. Thus, the allocated integer array uses half the storage required for a floating-point array of the same dimension. An integer array of dimension N uses N+1 words of storage. An integer array of dimension N by M uses N+1 times M+1 words of storage.

Storage for an integer array is allocated as the INTEGER statement is encountered. In standard memory systems, the size of an integer array is limited only by the amount of available free memory. **In extended memory (XM) systems, the maximum size of an integer array is limited to 16384, (16K) elements.** If there is not enough free memory available to contain the array, a fatal error is issued.

In TEK SPS BASIC, array indices are numbered from zero. An array A of dimension N has N+1 elements. The first element is A(Ø). The last element is A(N). Similarly, a matrix of dimension I,J has I+1 rows and J+1 columns for I+1 times J+1 elements. The first element in B is B(Ø,Ø) while the last is B(I,J).

Integer elements have a range of -32768 to +32767 (15 bits and a sign bit). When floating-point values are stored in an INTEGER array, the values are <u>truncated</u> to integers. (For example, 5.78 becomes 5 and -1Ø.2 becomes -1Ø.)

Integer and floating-point arrays may be used together in arithmetic expressions. However, TEK SPS BASIC does not do integer arithmetic. During expression evaluation, the elements of an integer array are temporarily converted to floating-point.

If you want an integer variable, but do not need an array, you can create an array with one element by using the INTEGER statement with the expression equal to zero. (Example: INTEGER A(Ø).) This results in a one element integer array. Future references to this array as a single variable must include the subscript of Ø.

Using the Syntax Options:

The simple numeric (not subscripted) variable is the name of the integer array after the command executes. If an array is used, it must be DELETEd if you are attempting to redimension it to new specifications. No error is issued if it is redimensioned to its current specifications.

The expressions in parentheses determine the size and number of dimensions (one or two) of the array. An expression is rounded to an integer and used as the largest index -- not the number of elements -- in a row or column of the array. Providing a single expression that evaluates to N allocates a one-dimensional array of N+1 elements. Supplying two expressions creates a two-dimensional array. If the expressions evaluate to M and N, a matrix of M+1 by N+1 elements is defined.

LET

Examples:

```
100 LET X = 5.7838
110 LET A(K) = X
112 LET B(15:30+J7) = B(5:20+J7)
114 LET X$ = "HELLO SPS"
120 B = P(0:100)*50/(X+Z)
130 A$(15) = "STRING 16"&BT$
170 Y = Y+1
```

Syntax Form:

```
                  ⎧⎧simple numeric variable⎫   ⎧expression           ⎫⎫
                  ⎪⎨array                  ⎬ = ⎨array expression     ⎬⎪
[line no.] [LET] ⎨⎩waveform               ⎭   ⎩waveform expression  ⎭⎬
                  ⎪array element = expression                          ⎪
                  ⎩string variable = string expression                 ⎭
```

Descriptive Form:

```
                  ⎧⎧target simple numeric variable⎫   ⎧expression           ⎫⎫
                  ⎪⎨target array                  ⎬ = ⎨array expression     ⎬⎪
[line no.] [LET] ⎨⎩target waveform               ⎭   ⎩waveform expression  ⎭⎬
                  ⎪target array element = expression                          ⎪
                  ⎩target string variable = string expression                 ⎭
```

Purpose:

To assign a value to a variable, string variable, array, waveform, array zone, array element, or waveform element.

Discussion:

The LET command is the workhorse of the Resident BASIC commands. This is the command that causes BASIC to create and assign value(s) to a new variable or auto-dimensioned array, or to change the value(s) of an existing variable. The variable, array, waveform, or string variable on the left of the equal sign (the target) is set equal to the value of the expression on the right of the equal sign (the source). Note that the equal sign does not mean "equal to" in the mathematical sense. Rather, it means "assign the value of the expression on the right to the variable on the left." Thus, the target will be equal to the source after the statement executes. For this reason a LET statement is also called an assignment statement.

The action of the LET command is in two steps. First the expression is evaluated. The expression, depending on its type, can result in a single number, an array of numbers, a waveform, or a single string. (The rules for expression evaluation are covered in Section 2 of this manual.) The second step is to assign the resultant value(s) to the target variable. How this is done depends on the type of both the target and the expression. The legal combinations are discussed below. A summary of the action resulting from the possible legal and illegal combinations are summarized in the table following the discussion.

Using the Syntax Options:

The word **LET** is optional. Statements that begin with a variable name and the equal sign are assumed to be LET statements.

When the target is a simple numeric variable and the source expression evaluates to a single value, that number is assigned to the variable. If the expression evaluates to an array or waveform, the variable is auto-dimensioned to a one-dimensional, floating-point array of the same size as the source array (or source waveform's array). Even when the source is a waveform, only an array is created -- not a waveform. This new auto-dimensioned array is set equal to the source array.

When the target is a floating-point array or array zone and the expression evaluates to a single value, every element in the array or array zone is set equal to that value. If the expression evaluates to an array or waveform, the target array (or array zone) and the source array (or the source waveform's array) must be the same size. The target array is set equal to the source array.

When the target is an integer array or integer array zone, the action is the same as for a floating-point array except that the source values are truncated before they are assigned.

When the target is a waveform, its array is assigned values by the rules for target arrays, explained above. If the expression evaluates to a waveform, the target waveform's data sampling interval (DSI) and horizontal and vertical units are set to those of the source waveform. If the expression results in a single value or an array, the target waveform's DSI is set to zero and its units are set to null strings. This is true even if the target waveform's DSI or units had been previously assigned.

When the target is an array element, only one value can be legally assigned to it. Unless the expression evaluates to a single number, an error is issued. If the target is an integer array element, the source value is truncated to an integer when it is assigned.

When the target is a string variable, the expression must evaluate to a string. Notice that a string array may not be the target variable. **A single string array element may be referenced by its subscript, but string arrays, unlike numeric arrays, cannot be filled with one LET statement.**

Summary of LET Command

Target Variable	Result of Source Expression	Action
simple numeric variable	single number	target set equal to number
	array or waveform	target auto-dimensioned, then set equal to source array (or waveform's array)
	string	error
floating-point array or waveform element	single number	target set equal to number
	array, waveform, or string	error
integer array or waveform element	single number	target set equal to truncated number
	array, waveform, or string	error
floating-point array or array zone	single number	each element of target set equal to the number
	array or waveform	arrays (or array zones) must be same size, else error; target set equal to source array (or waveform's array)
	string	error

Summary of LET Command, cont.

integer array or array zone	single number	same as for floating-point array but number truncated
	array or waveform	same as for floating-point array, but values truncated
	string	error
waveform	single number	each element of target's array set equal to the number; target's DSI set to zero; target's units set to null
	array	target's array and source must be same size, else error; target's array set equal to source; target's DSI set to zero; target's units set to to null
	waveform	associated arrays must be same size, else error; target's array, DSI, and units set equal to source's array, DSI, and units.
	string	error
string variable	single string	target set equal to source string
	string array	error
	numeric value, array, or waveform	error

LIST

Examples:

```
    LIST LP:,1ØØ,5ØØ
    LIST
1ØØ LIST DK1:"PRGM.LST",1Ø,999
    LIST DX:A$
    LIST 2Ø
```

Syntax Form:

```
[line no.] LIST ⎡expression[,expression]                                                          ⎤
                ⎣device name[constant]:[string expression][,expression[,expression]]⎦
```

Descriptive Form:

```
[line no.] LIST ⎡line number [starting,line number ending]                        ⎤
                ⎢device name [drive number]:[target file name]                    ⎥
                ⎣            [,line number[starting,line number ending]]⎦
```

Purpose:

To list all or part of the current program text on the terminal or the specified peripheral.

Discussion:

This command allows you to look at all or part of the program text currently in memory. The text is printed on the terminal (the default device) or another peripheral, such as a line printer.

When a program is listed, control characters are printed as the ASCII letter preceded by an up-arrow (^). Because of this, programs LISTed to a storage peripheral (such as a disk), cannot later be brought into memory via the OLD, CHAIN, or OVERLAY commands. (The SAVE or REPLACE command must be used to output a program that will be read into memory again by the OLD command.) A program that has been LISTed to a file can be displayed later by COPYing the file to the keyboard terminal (KB) or a line printer (LP). But remember, the purpose of LIST is to display, not to SAVE your program.

If a line to be listed contains more than 8Ø characters, the entire line is printed and a warning error is issued. (The error is to tell you that if you SAVE the line, it cannot be loaded again in its entirety.)

If a program containing a LIST statement is RENUMbered, the expressions in the LIST command for line numbers are **not** altered.

Using the Syntax Options:

The named device is the peripheral to which the listing is sent. If the device is omitted, the terminal is assumed. If the device is not the system device or the terminal, its driver must be LOADed into memory before the command is executed. If the drive number is omitted, zero is used.

A file name must be supplied with a file-structured device. A file with that name must not already exist on the designated device.

The line numbers define the scope of the lisitng. If no line number is present in the LIST statement, all program lines are printed. If only one line number is given, only that line is listed. When two line numbers appear in the command, all lines between and including those lines are listed.

If there is no text which meets the given line number conditions, nothing is output. If the target device is file-structured, no file is created.

LISTVAR (Nonresident)

Examples:

```
   LISTVAR
45 LISTVAR DK:"LIST.FIL"
65 LISTVAR LP:
```

Syntax Form:

[line no.] **LISTVA** [device name[constant]:[string expression]]

Descriptive Form:

[line no.] **LISTVAR** [device name[drive number]:[target file name]]

Purpose:

To list on the terminal or to a specified file or peripheral the names and dimensions of all arrays, waveforms, variables, string variables, and string arrays that are in memory.

Discussion:

LISTVAR allows you to see what is currently defined in memory. Every variable, array, waveform, string variable, or string array name is listed, either at the terminal or on a specified device.

When the LISTVAR output is directed to a file, it can later be printed on the terminal by COPYing the file to the keyboard terminal (KB). Also, it can be read back into memory by INPUTting that file one line at a time as a string variable.

Using the Syntax Options:

If no device is specified, output goes to the terminal. Otherwise, output is directed to the specified device. If the named device does not use the system device driver, its driver must be in memory before LISTVAR executes. If the device is file-structured, a file name must be included in the command. The named file must not already exist on the device. When the device number is omitted, zero is assumed.

Output Format:

The format of the output is explained below. Here XX stands for a variable name, ZZZ stands for an array dimension, and spaces that always appear are shown by a dashed underscore (_).

1) Floating-point arrays:
   __XX(ZZZ,ZZZ)_
   Five array names are printed per line with 14 characters per each array. Each array specified includes two spaces (or three spaces if the name has only one letter), name, left parenthesis, first dimension, comma and second dimension (if nonzero), right parenthesis, and spaces to pad to 14 characters.

2) Integer arrays:
   Same format as floating-point arrays.

3) Waveforms:
   __XX____IS__XX,_XX,_XX$,_XX$
   One waveform is printed per line with 28 characters per waveform. The parts of the waveform correspond to the parts and sequence in the WAVEFORM command. The format is two spaces (or three spaces if the waveform name has only one letter), waveform name, four spaces, keyword IS, two spaces (or three spaces if array name has only one letter), array name, comma, one space (or two spaces if the name of the data sampling interval variable has only one letter), the DSI variable name, comma, one space (or two spaces if the name of the horizontal units variable has only one letter), horizontal units variable name with a dollar sign, comma, one space (or two spaces if name of vertical units variable

has only one letter), and vertical units variable name with a dollar sign.

4) Numeric variables:
__XX__
Up to 12 variable names are printed per line with six characters per each variable. The format is two spaces (or three spaces if the name has only one letter), name, and two spaces.

5) String variables:
__XX$_
Up to 12 string variable names are printed per line with six characters per each variable. The format is two spaces (or three spaces if the name has only one letter), name with a dollar sign, and one space.

6) String arrays:
__XX$(ZZZ,ZZZ)
Up to five array names are printed per line with 14 characters per each array. The specification of a string array includes two spaces (or three spaces if the name has only one letter), name, with a dollar sign, left parenthesis, first dimension, comma and second dimension (if nonzero), right parenthesis, and spaces to pad to 14 characters.

Output Example:

The following is an example of the LISTVAR output.

```
FLOATING POINT ARRAYS
   A(55)         B(1Ø,5)      JA(511)       KA(511)

INTEGER ARRAYS
   C(5ØØ)

WAVEFORMS
   J    IS  JA, DJ, HJ$, VJ$
   K    IS  KA, DK, HK$, VK$
```

```
NUMERIC VARIABLES
  DJ    DK

STRING VARIABLES
  HJ$   HK$   VJ$   VK$

STRING ARRAYS
  ST$(4Ø)
```

LOAD

Examples:

```
100 LOAD "PP.SPS","USER.SPS"
150 LOAD PR:"GRAPH"
    LOAD CT:/F,GR$
```

Syntax Form:

[line no.] **LOAD** [device name[constant]:] $\left[/ \begin{Bmatrix} \mathbf{F} \\ \mathbf{R} \end{Bmatrix} [,] \right]$ string expression
[,string expression] ...

Descriptive Form:

[line no.] **LOAD** [device name[drive number]:][/ forward or reverse switch [,]]
driver or command name [,driver or command name] ...

Purpose:

To bring nonresident commands and instrument and peripheral drivers into the controller memory from a peripheral device.

Discussion:

The LOAD command is the heart of SPS modularity. Together with the RELEASE command, LOAD allows you to "customize" controller memory to give you the most free memory space for a particular job. You may LOAD a nonresident command or driver as you needed it and RELEASE it when done with it.

Nonresident drivers cannot be auto-loaded. They must be explicitly brought into memory by the LOAD command. Nonresident commands, however, may be either explicitly LOADed or auto-loaded. When a driver or command has been brought into memory by the LOAD command, it stays resident until released with the RELEASE command. The LOAD command "locks in" modules;

auto-loaded commands are released by BASIC whenever room is needed for program text, arrays, drivers, or other nonresident commands.

Once nonresident commands are loaded, they can be used in programs in the same manner as resident commands. LOADing frequently used commands can shorten program execution time.

If the specified device is a serial access, file-structured device such as cassette tape, no file may be open on it when LOAD executes.

Using the Syntax Options:

The system storage device is used by the LOAD command as the source peripheral device if no other device is specified. If the named device is not the system device, its driver must be LOADed into memory before a command or driver stored on it can be LOADed. If the drive number is omitted, zero is assumed.

[When a serial tape device is specified, the **/F** or **/R** (Forward or Reverse) switches may be used. If a switch is not present, the tape is rewound before a forward search for the file begins. If the device is not serially structured, the /F or /R switch is ignored. If the end of tape is reached before a specified file is found, a fatal error is issued.]

The LOAD command will only load files with .SPS extensions. The extension need not be specified but if it is, it must be .SPS. (The .SPS extension is reserved for the file names of nonresident commands and drivers.

[When the specified device is paper tape, any unused command name may be supplied. That name is then associated with the file loaded. When paper tape is used, the correct tape must be in the tape reader before executing the LOAD command.]

LOCKKB (Nonresident)

Examples:

```
 1Ø LOCKKB
9ØØ LOCKKB OPEN
```

Syntax Form:

[line no.] **LOCKKB** [**OPEN**]

Purpose:

To limit system input to a Control-P while a program is running.

Discussion:

Normally the keyboard is "live". You can enter a line of characters (terminated by a carriage return) from the keyboard at any time during a running program. If an input request condition is enabled by the INPREQ command or an INPUT statement is waiting for data, this line is interpreted to be program data. Otherwise this line is interpreted to be system input -- either a new line of program text or an immediate mode command. However, after a LOCKKB is executed, the keyboard is locked and the only system input accepted while a program is running is a Control-P. (Of course, system input is still accepted when no program is running.)

With the keyboard locked, input from the keyboard (while a program is running) is only accepted if prompted by an INPUT command or allowed by an INPREQ command.

A locked keyboard can only be made live again by executing a LOCKKB OPEN statement. END, STOP, and Control-P will **not** restore the keyboard to its normal (live) condition.

Using the Syntax Options:

When just the command name **LOCKKB** is used, the keyboard is locked. When the keyword **OPEN** is specified, the keyboard is restored to normal.

LST (Nonresident)

Examples:

```
525 LST LP:,1ØØ,2ØØ
    LST
    LST 2Ø
76Ø LST DK1:"LST.FIL",1ØØ,2ØØ
    LST DX:A2$
```

Syntax Form:

```
                  ┌expression[,expression]                                                    ┐
[line no.] LST    └device name[constant]:[string expression][,expression[,expression]]        ┘
```

Descriptive Form:

```
                  ┌line number[starting, line number ending]                   ┐
[line no.] LST    │device name[drive number]: [target file name]               │
                  └          [,line number [starting, line number ending]]     ┘
```

Purpose:

To display all or part of the current program text on the terminal or the specified peripheral device with indented FOR/NEXT loops.

Discussion:

The LST command is like the LIST command except for the way the program lines are output. The line numbers are right-justified in a five-character field and statements that were concatenated with a backslash (\) are output on separate lines. Also, statements within a FOR/NEXT loop are indented two print positions from the position of the FOR statement. When FOR/NEXT statements are nested, the inner loops are indented from the outer loops. For example, LIST displays a program like this:

```
*LIST
10 FOR I=1 TO 10
200 B=0
3000 FOR K=1 TO 1000
4000 C(K)=1 \ FOR J=1 TO 10\A(J)=1\NEXT J
10000 NEXT K
20000 NEXT I
```

while, LST displays the same program like this:

```
*LST
   10 FOR I=1 TO 10
  200   B=0
 3000   FOR K=1 TO 1000
 4000     C(K)=1\
          FOR J=1 TO 10\
            A(J)=1\
          NEXT J
10000   NEXT K
20000 NEXT I
```

If a section of a program is listed by LST, the indentations of any statements within FOR/NEXT loops reflect the nesting from the beginning of the program. For instance, LST displays lines 4000 through 20000 of the example program this way:

```
*LST 4000,30000
 4000     C(K)=1\
          FOR J=1 TO 10\
            A(J)=1\
          NEXT J
10000   NEXT K
20000 NEXT I
```

See the LIST command for more information.

NOTE

The LST command is not available with TEK SPS BASIC VØ2-Ø1.

MATCH (Nonresident)

Examples:

```
100 MATCH A$,B$,N
150 MATCH AR$,"THE",K,M
170 MATCH R$,W$&".MAC",W(N),W(N+2)
```

Syntax Form:

[line no.] **MATCH** string array,string expression,variable[,variable]

Descriptive Form:

[line no.] **MATCH** string array, search string, target variable for array index
[,target variable for search string's starting position]

Purpose:

To search a string array for a given search string.

Discussion:

The MATCH command searches a string array (the first argument) for the first occurrence of the search string (the second argument). A match is considered found if an element of the string array completely contains the search string, either as an absolute match or as a substring. If a match is found, the index of the element containing the search string is returned in the third argument. If a match is not found, negative one (-1) is returned instead. Optionally, the starting position of the search string within the matching array element is also returned.

If the array is two-dimensional, the value returned in the third argument is equal to the element's first subscript times the quantity, one plus the maximum value of the array's second subscript, plus the elements second subscript. For example, if a match is found in element A(I,J) of an array A dimensioned M by N, the third argument is equal to I*(1+N)+J.

Using the Syntax Options:

The first argument is the string array to be searched. Remember, string arrays cannot be zoned. The second argument, a string expression, is the search string.

If a match is found, the index number of the first array element containing the search string is returned in the third argument, a variable. When no match is found, it will contain a -1.

The optional fourth argument is also a variable. When present, the search string's starting position within the string element is returned in it. As with the third argument, when no match is found, it will contain a -1.

Application Example:

The match command can be used to check for a proper input string when the returned index is used in a computed GOSUB or GOTO statement. In the example below, the days of the week are entered into the string array, D$. Then the program asks for a day name to be INPUT. Line 2ØØ compares the input string with the days of the week. Depending on which day is entered, a specific subroutine is jumped to via a computed GOSUB statement (line 21Ø). If there is no match, the -1 that is returned in N sends the program back to line 18Ø to ask again for the INPUT. Otherwise, program control transfers to the desired subroutine.

```
100 DIM D$(6)
110 D$(0)="SUNDAY"
120 D$(1)="MONDAY"
130 D$(2)="TUESDAY"
140 D$(3)="WEDNESDAY"
150 D$(4)="THURSDAY"
160 D$(5)="FRIDAY"
170 D$(6)="SATURDAY"
180 PRINT "ENTER THE DAY OF THE WEEK"
190 INPUT S$
200 MATCH D$,S$,N
210 GOSUB N+2 OF 180,1000,2000,3000,4000,5000,6000,7000
 .
 .
 .
```

Notice that since a match is recognized when the search string is a substring as well as an exact match, entering the first three letters as an abbreviation works just as well as entering the entire name. However, just an "S" will always match D$(Ø) and never D$(6).

NEXT

Example:

```
100 NEXT J
```

Syntax Form:

[line no.] **NEXT** simple numeric variable

Descriptive Form:

[line no.] **NEXT** index

Purpose:

To delimit the scope of a FOR/NEXT loop.

Discussion:

The NEXT command is the last statement in a FOR/NEXT program control loop. See the FOR command documentation.

Using the Command Syntax:

The simple numeric variable is the loop index. The NEXT command and its corresponding FOR command must use the same variable name.

OLD

Examples:

```
    OLD "DEMO.BAS"
150 OLD PR:
160 OLD DK2:A$
170 OLD CT1:/R,"SAMPLE",200
```

Syntax Form:

$$\text{[line no.] } \textbf{OLD} \text{ [device name[constant]:]} \left[/ \begin{Bmatrix} \text{F} \\ \text{R} \end{Bmatrix} \text{ [,]} \right] \text{[string expression][,line number]}$$

Descriptive Form:

```
[line no.] OLD [device name[drive number]:][/ forward or reverse switch [,]]
               [program file name][,line number where execution starts]
```

Purpose:

To bring a BASIC program from the specified device into memory.

Discussion:

Programs that you have previously SAVEd on a peripheral device are brought back into the controller memory with this command.

When the OLD command is executed, all currently defined variables, arrays, waveforms, string variables, and string arrays are deleted, along with any existing program lines in memory. Files are left OPEN, but actions of all WHEN, SCHEDULE, INPREQ, and ONERR commands are cancelled and the Scheduler is cleared. (The function and parts of the Scheduler are explained in Section 1.) Then the named BASIC program is brought into memory. If the OLD command encounters a line of text longer than 8Ø characters, a warning error is issued, and the extra characters are dropped. Since the OLD command deletes all program text in memory, an OLD statement should not be followed by a backslash (\).

When a line number is specified, execution automatically begins at the line number (or the next higher line number if the specified one does not exist). For example, in statement 170 in the example above, execution would begin at line 200 in the program "SAMPLE".

[The new program executes with a task number equal to the task number of the OLD statement, unless the OLD statement is in the immediate mode. In which case, the task number is set to zero. Thus, the immediate mode command

```
OLD "PROGRM",1
```

causes "PROGRM" to execute as task zero, not 127 which is the immediate mode task number.]

If the line number is omitted, what happens depends on whether the OLD command is issued in program mode or immediate mode. In program mode, execution continues with the first line of the new program. [Its task number is equal to the task number of the OLD statement, except when that task number is 127. Then the task number of the new program is zero.] In immediate mode, the next command entered in immediate mode (after the OLD command executes) is executed.

To execute the new program, a RUN or GOTO (with a line number) can be entered in immediate mode. [Starting the program with RUN makes its task number zero unless the task number is explicitly stated. However, starting the program with an immediate mode GOTO makes its task number 127, the immediate mode task number.]

For other commands that bring a stored program into memory see CHAIN, OVERLAY, and OVLOAD.

Using the Syntax Options:

When no device is specified, the system device is used. If the named device does not use the system device driver, its driver must be LOADed into memory before the OLD command is executed. The keyboard (KB) may not be specified. If the device is a serial-access, file-structured device, no files may be OPEN on it when OLD executes. If no drive number is specified, zero is assumed.

[The Forward or Reverse switches (**/F** or **/R**) may be included in the command if the peripheral device is a serial tape device. These switches specify the direction of tape movement when searching for the file. If the switch is omitted, the tape is rewound before the search for the file is made. If the device is not a serial tape device, a **/F** or **/R** switch is ignored.]

A file name must be specified for a file-structured device. If no extension is present in the file name, .BAS is assumed.

If a line number is given, it must be an integer between 1 and 32767, inclusive. What happens when it is included or omitted is explained above.

ONERR (Nonresident)

Examples:

```
150 ONERR IA GOTO 750
970 ONERR RETURN
890 ONERR RETURN GOTO 500
100 ONERR NOWARN
500 ONERR
```

Syntax Form:

```
                     ┌ ┌integer array┐                  ┐
                     │ │             │ GOTO line number │
[line no.] ONERR     │ └variable     ┘                  │
                     │ RETURN [GOTO line number]        │
                     └ NOWARN                           ┘
```

Descriptive Form:

```
                     ┌ target for error information  GOTO line number ┐
[line no.] ONERR     │ RETURN [GOTO line number]                       │
                     │ NOWARNing error messages                        │
                     └                                                 ┘
```

Purpose:

To allow the detection and handling of warning and fatal errors in a BASIC routine.

Discussion:

Usually when an error occurs, an error message is displayed on the terminal and, if the error is fatal, the task associated with it halts. However, the ONERR command lets you process errors in your own way instead of being forced to use BASIC's built-in error procedures. (These procedures are outlined in "Understanding Errors", Section 8.)

With the ONERR GOTO form of the command you can write your own error-handling routines. The error message is not displayed and, in most cases, even if the error is fatal, the task committing the error is not aborted. Instead, program control transfers to the specified line number, which should be the first line of your error-handling routine.

[When an error occurs with an ONERR GOTO condition active, the specified line number is entered in the Scheduler queue with a priority of 127 and a task number equal to the current task number. Since the entered line number has the highest priority possible, it becomes the current task as soon as control returns to the Scheduler. Statements that only cause warning errors finish executing before the user-written error handler is jumped to via the Scheduler. Also, when errors occur during the execution of an I/O command, the input or output finishes before control passes to the Scheduler and on to the error routine. Otherwise, commands with fatal errors do not complete execution and control returns to the Scheduler after the line number is entered.]

[The ONERR GOTO form of the command prevents most fatal errors from triggering the reset actions that halt a task's execution as outlined in the section on errors. However, if the error is caused by an overflow of the Scheduler stack or queue, a system reset is performed. The ONERR is disabled so the program halts.]

ONERR cannot be used to handle a peripheral hardware I/O (P18) error. If a P18 error occurs, any ONERR condition is ignored and the error is handled as if an ONERR were not set up, but the ONERR remains enabled.

If more than one ONERR GOTO statement appears in a program and an error occurs, program control transfers to the line number specified in the most recently executed ONERR GOTO.

After an error is detected, the target array contains the information about the error normally printed in the error message. The contents of the array elements have the following meanings:

Index	**Content and Meaning**
element Ø	the line number of the command where the error occurred (or Ø if in immediate mode)
element 1	The decimal equivalent of the ASCII character in the error code
element 2	The numeric portion of the error code
element 3	A Ø or 1 if a fatal error; a 2 if a warning error

If a fatal error occurs while the user-written error-handling routine is executing, the ONERR is disabled. The new error is handled as if an ONERR was not set up and the task is aborted. If a warning error occurs during the user's error routine, a warning message is printed and execution continues.

The error-handling routine is terminated by a form of the ONERR RETURN statement. Program control can either be returned to the statement immediately following the one that caused the error or be transferred to a specified line number. In either case, the system priority returns to the priority of the statement that caused the error. If a form of the ONERR RETURN command is executed when no error has occurred, an error is issued.

You should use some form of the ONERR RETURN statement to exit the error routine. If a regular GOTO statement is used, the system priority remains at the priority of the error-handling routine, which is initially set to 127. Also, the system will assume that the error routine is still executing. Another error will either disable the ONERR or cause an error message to be printed, as explained above.

If all you want to do is suppress the printing of warning error messages and nothing else, use the ONERR NOWARN form of the command. It does not cause transfer of control when an error occurs. If a transfer was set up by a previously executed ONERR GOTO statement, that transfer is **not** disabled.

To disable all ONERR conditions set up by previously executed ONERR statements, execute the ONERR command using no keywords. Control-P, END, STOP, OLD, CHAIN, DELETE ALL, and DELETE TEXT also disable any ONERR conditions. ONERR is not disabled when an error occurs unless the error is fatal and the user-written error-handling routine is executing.

Using the Syntax Options:

The keyword **GOTO** sets up a transfer to the specified line number, should an error occur. Either a simple numeric variable or an integer array is used to return the information about the error. If a simple numeric variable is specified, it is auto-dimensioned to a four-element integer array. If an integer array is specified, it must contain exactly four elements.

The keyword **RETURN** terminates a user-written error-handling routine. This form of the ONERR command must appear only in the error-handling routine. If the optional keyword **GOTO** followed by a line number is used also, program control passes to the specified line number. If the GOTO and line number are omitted, program control returns to the statement following the command that committed the error.

The keyword **NOWARN** disables the display of warning error messages on the terminal.

When the **ONERR** command name is used alone (omitting all keywords), any previously executed ONERR conditions are disabled.

Application Example:

Suppose a program needs a particular instrument to be ATTACHed, but doesn't know what instrument logical unit numbers (ILUNs) are free. It may not even know if the instrument is already ATTACHed. Trying to ATTACH an instrument to a ILUN that is not free causes an I3 error. Attempting to ATTACH an already attached instrument to a different ILUN is an I15 error. Normally, either error is fatal, but with the ONERR command, you can write a routine to handle both possibilities. To help you understand how to use ONERR, consider this over-simplified solution.

```
10 REM SET UP ONERR TRANSFER
20 ONERR AR GOTO 500
30 J=1
```

```
40 K=J
50 ATTACH #J AS DPO1:
60 IF J<>K THEN 40
70 REM DISABLE ONERR TRANFER
80 ONERR
 .
 .
 .
490 REM IGNORE IF IMMEDIATE MODE ERROR
500 IF AR(0)=0 THEN 650
510 REM TEST FOR EXPECTED LINE NUMBER
520 IF AR(0)<>50 THEN 590
530 REM TEST FOR INSTRUMENT ERROR
540 IF AR(1)<>ASC("I") THEN 590
550 REM CHECK FOR I3 or I15 ERROR
560 IF AR(2)=3 THEN 640
570 IF AR(2)=15 THEN 640
580 REM IF NOT I3 OR I15 ERROR, WRITE MESSAGE
590 PRINT CHR(AR(1));AR(2);" ERROR IN LINE";AR(0)
600 REM IF WARNING ERROR, RETURN; IF FATAL, ABORT
610 IF AR(3)=2 THEN 650
620 ONERR RETURN GOTO 670
630 REM INCREMENT ILUN AND TRY AGAIN TO ATTACH
640 J=J+1
650 ONERR RETURN
670 ABORT
```

Starting with the ILUN variable J set to 1 and a test variable K set equal to J, the program tries to ATTACH DPO1. If it works, K still equals J so the program continues. If ATTACH fails, control transfers to the error routine which expects an I3 or I15 error in line 50. By checking the elements of the array AR, the routine determines if one of the expected errors triggered the jump. Any other program error in any other line is reported and, if fatal, ABORTS the current task (line 670). (Immediate mode errors are ignored.)

If the error is as expected, the trial ILUN, J, is incremented and control passes back to the main program (line 60). Since J no longer equals K, the program jumps back to line 40. K is reset to J and another attempt is made to ATTACH DPO1. This loop continues until a free ILUN is found, the ILUN already ATTACHed to DPO1 is matched, or the legal range of ILUNs is exceeded. That last possibility would ABORT the current task because this error routine cannot recover from that or any other fatal error.

OPEN

Examples:

```
100  OPEN #4 AS PR: FOR READ
110  OPEN #2 AS LP: FOR WRITE
120  OPEN #1 AS CT:/F,"CASSET.FIL" FOR READ
130  OPEN #K+2 AS DK:FL$ FOR WRITE WITH N INTO 10
140  OPEN #N AS DX1:"RECORD.DAT" FOR UPDATE
150  OPEN #3 AS DK1:"LASER.001" FOR WRITE INTO 6
```

Syntax Form:

$$[\text{line no.}]\ \textbf{OPEN}\ \textbf{\#}\text{expression}\ \textbf{AS}\ [\text{device name}[\text{constant}]\textbf{:}]\left[/\left\{\begin{matrix}\textbf{F}\\ \textbf{R}\end{matrix}\right\}[\textbf{,}]\right][\text{string expression}]$$

$$\textbf{FOR}\left\{\begin{matrix}\textbf{READ}\\ \textbf{WRITE}\ [\textbf{WITH}\ \text{expression}]\ [\textbf{INTO}\ \text{expression}]\\ \textbf{UPDATE}\end{matrix}\right\}$$

Descriptive Form:

$$[\text{line no.}]\ \textbf{OPEN}\ \textbf{\#}\text{plun}\ \textbf{AS}\ [\text{device name}[\text{drive number}]\textbf{:}]$$
$$[/\text{forward or reverse switch}[\textbf{,}]][\text{file name}]$$
$$\textbf{FOR}\left\{\begin{matrix}\textbf{READ}\\ \textbf{WRITE}\ [\textbf{WITH}\ \text{number of buffers}]\ [\textbf{INTO}\ \text{number of blocks}]\\ \textbf{UPDATE}\end{matrix}\right\}$$

Purpose:

To allow access to an existing data file, a new data file, or a non-file-structured peripheral device in order for the input or output of data to take place.

Discussion:

The OPEN command makes a data file or device accessible for the input or output of data by associating it with a peripheral logical unit number (PLUN). Once the PLUN is assigned, the file or the device is referenced by that number rather than by its name.

A PLUN may not be associated with a file already OPENed nor may the same PLUN be assigned to more than one file or device at a time. A PLUN must be freed by a CLOSE or END statement before it can be reassigned.

The terminal keyboard is permanently assigned to PLUN zero and is always OPEN for both READ and WRITE. PLUN zero may not be assigned to any other device or file. The keyboard may not be associated with another PLUN.

Using the Syntax Options:

In the command syntax, the PLUN is preceded by the pound sign (#). Since the PLUN may be an expression, if it does not evaluate to an integer, it is rounded to an integer value. The number of PLUNs that may be assigned at any one time, n, is determined at system initialization. The value assigned as a PLUN must be an integer between 1 and n, inclusive. The default number of PLUNs is six. In this case, the numbers 1 through 6 may be assigned as PLUNs. (To change the number of PLUNs allowed see the SYSBLD command.)

The device name follows the keyword **AS**. When no device name is specified, the system device is assumed. If the named device does not use the system device driver, the driver for that device must be LOADed into memory before OPEN executes. When the device named is a serial tape device, only one file can be OPENed on it at a time. If the drive number is omitted, zero is used.

[The optional forward or reverse switch **/F** or **/R** is for use with a serial tape device being OPENed FOR READ. It causes a search of the tape in a forward or reverse direction, respectively. If the switch is omitted, the tape is rewound before a forward search is begun. The search ends when the file is found or an end-of-tape is reached. If the device is not a serial tape device being OPENed FOR READ, the /F or /R is ignored.]

The next argument in the command syntax is the file name. The file name must be included if the specified peripheral is a file-structured device such as disk or magtape. The file name is optional when the device is a non-file-structured device such as a line printer or a paper-tape reader or punch. (If a file name is included in a statement to OPEN a paper-tape punch FOR WRITE, the file name is punched on the tape in humanly-readable alphanumeric characters.) The OPEN command has no default file name extension.

The OPEN command determines the mode of access to receive or send data (random or sequential) by the keywords UPDATE, READ, or WRITE.

For random access, a file is OPENed as a record I/O file by using the keyword **UPDATE**. The peripheral must be a directory-structured device, and the file must already exist on it. (See the DEFINE command for information on creating a record I/O file.) Once OPENed FOR UPDATE, a file is accessed for input or output by using the record I/O form of the READU or WRITEU commands, respectively.

To receive data in a sequential manner from a peripheral device or file you use the keyword **READ**. If a file name is given, that file must already exist on the device. Files or devices OPENed for READ are accessed for input only by the READ, INPUT, or regular form of the READU commands.

The keyword **WRITE** is used to send data to a peripheral in a sequential manner. If the device is file-structured, a file is created. A file of the same name must not already exist on the device or a fatal error results. If the specified device is a paper-tape punch, leader is punched. (When the punch is closed with the CLOSE or END command, leader is again punched.) Files or devices OPENed for WRITE are accessed for output only by the WRITE, PRINT, or regular form of the WRITEU commands.

[You can increase the speed of throughput to a file or device OPENed FOR WRITE by using more than one buffer. Multiple buffers are specified by the expression following the optional keyword **WITH**. The expression is rounded to an integer if necessary. Using more than one buffer requires additional memory for each extra buffer. See the Peripheral Drivers manual for buffer sizes. If this option is omitted, a default of one buffer is used.]

[The expression following the keyword **INTO** allows you to specify the maximum size in blocks (256 words per block) of a file OPENed FOR WRITE on a directory-structured device. The result of the expression is rounded to an integer if necessary. If this option is omitted, half of the largest, contiguous free space on the device is used as the new file.]

[The file created by an OPEN for WRITE statement may be smaller than the allotted or default space, but it cannot be larger. As your disk begins to fill, the default (half the available space) may not be large enough, even for small files. In order to use more than half of the largest, contiguous free space for a file, you need to include the INTO specification when OPENing a file FOR WRITE. If the allotted or default space exceeds the actual number of blocks used by the file, the extra blocks are returned to a free status when the file is CLOSEd.]

OVERLAY (Nonresident)

Examples:

```
100 OVERLAY DK1:"PROG2.BAS"
120 OVERLAY "NEXT"
150 OVERLAY CT:/R,B$
160 OVERLAY PR:
```

Syntax Form:

[line no.] **OVERLA** [device name[constant]:] $\left[/ \left\{ \begin{matrix} \mathrm{F} \\ \mathrm{R} \end{matrix} \right\} [,] \right]$ [string expression]

Descriptive Form:

[line no.] **OVERLAY** [device name[drive number]:][/ forward or reverse switch [,]]
[program file name]

Purpose:

To move a new program segment into memory, overlaying an existing program segment. Execution continues if the command is executed in program mode.

Discussion:

When very large programs are written, it is often desirable to break them into several smaller segments and execute each segment one at a time. The OVERLAY command allows this flexibility.

If the OVERLAY command is used in a program (not in immediate mode), execution of the overlaid program is automatic. In this case, a portion of the current program is overlaid by the program segment stored in the specified file by a SAVE or REPLACE statement. Statements in the new program whose line numbers match existing statements in memory replace those existing lines. However, unlike CHAIN, OVERLAY deletes no other text. Execution resumes with the next sequential statement following the OVERLAY command. No variables are deleted, and files are left open. **The line containing the OVERLAY command must not be overlaid.**

All lines read in with the OVERLAY command must have line numbers. If a line contains more than 8Ø characters, a warning error is issued and the remainder of the line is ignored.

For other commands that allow efficient use of memory space see the discussions on CHAIN, GOSUB, and OVLOAD.

Using the Syntax Options:

When no device is specified, the system device is used. If the named device does not use the system device driver, its driver must be LOADed into memory before the OVERLAY command is executed. The keyboard (KB) may not be the specified device. If the drive number is omitted, zero is assumed.

[The **/F** or **/R** switches (Forward or Reverse) may be specified for a serial tape device. The switch indicates the direction of the tape movement when searching for the file. If the switch is omitted, the tape is rewound before a forward search is begun. The search stops when the file is found or an end of tape is reached. When used with other peripherals, the switch is ignored.]

A file name must be specified if the device is file-structured. If an extension is not included in the file name, .BAS is assumed.

OVLOAD (Nonresident)

Examples:

```
150 OVLOAD "PART2"
780 OVLOAD DK1:A$
170 OVLOAD CT:/R,"SECTN.3"
```

Syntax Form:

[line no.] **OVLOAD** [device name[constant]:] $\left[/ \left\{ \begin{matrix} \mathrm{F} \\ \mathrm{R} \end{matrix} \right\} [,] \right]$ string expression

Descriptive Form:

[line no.] **OVLOAD** [device name[drive number]:][/ forward or reverse switch [,]]
file name of pretranslated text

Purpose:

To perform a fast overlay of a pretranslated BASIC program segment from a file created by an OVLSAV statement.

Discussion:

The OVLOAD command allows faster execution of overlaid program segments than the OVERLAY command. The program text brought into memory by OVERLAY must be translated as it is loaded. However, the program text loaded by OVLOAD has been stored in a translated form by an OVLSAV statement.

Before the fast overlay file is loaded, any text in memory with line numbers in the range of the line numbers in the fast-overlay file are deleted. The line containing the OVLOAD command must not be in this range. Because the text is deleted, you cannot use an interleaved overlay technique with OVLOAD as you can with OVERLAY. (OVERLAY overwrites lines but does not delete lines.)

When the fast overlay file is loaded, there must be enough free memory available for one input/output buffer, the translated text, and any extra information about the text from the file. The size of the I/O buffer depends on the type of the device the file is stored on. See the Peripheral Drivers Manual for buffer sizes.

NOTE

A fast-overlay file created by the standard memory version of OVLSAV cannot be brought into memory by the extended memory (XM) version of OVLOAD, and vice versa.

Using the Syntax Options:

The named device must be file-structured. When no device is specified, the system device is used. If the named device does not use the system device driver, its driver must be LOADed into memory before the OVLOAD command executes. If the drive number is omitted, zero is assumed.

[The **/F** or **/R** switches (Forward or Reverse) may be specified for a serial tape device. The switch indicates the direction of the tape movement when searching for the file. If the switch is omitted, the tape is rewound before a forward search is begun. The search stops when the file is found or an end-of-tape is reached. When used with other peripherals, the switch is ignored.]

The string expression is the name of the fast overlay file. If an extension is not included in the file name, ".BOL" is assumed.

OVLSAV (Nonresident)

Examples:

```
450 OVLSAV A$
    OVLSAV DK1:"PART23.V11",1000,1500
    OVLSAV DX1:"NEXT"
    OVLSAV DL2:"FAST2" INTO 5
```

Syntax Form:

```
[line no.] OVLSAV [device name[constant]:]string expression [INTO expression]
                  [,expression[,expression]]
```

Descriptive Form:

```
[line no.] OVLSAV [device name[drive number]:] file name for fast-overlay file
                  [INTO number of blocks][,line number [starting, line number ending]]
```

Purpose:

To create a file containing a pretranslated BASIC program segment that can be loaded into memory by an OVLOAD statement.

Discussion:

As a BASIC program is entered from the keyboard terminal, the text is translated into an internal form and stored in the controller memory. Whenever a program in memory is displayed by a LIST statement or stored on a peripheral by a SAVE or REPLACE statement, the program is converted back to the familiar BASIC language form. This means that when a stored program is brought into memory by an OLD, CHAIN, or OVERLAY statement, time is required to translate the text back into the internal form. In large, heavily overlaid programs this translation time can represent much of the total execution time. OVLSAV provides the ability to save portions of a BASIC program in its already translated (internal) form. When program segments saved by OVLSAV are later loaded into memory by an OVLOAD statement, the translation process is completely avoided and the time saving is considerable.

Like SAVE, the OVLSAV can create a file from all or part of the program text that is in controller memory. Specifying one or two optional line numbers allows you to save in the given file only part of the program text that is in memory. If there is text in memory in the range of the line numbers specified, first, any file with the given file name is canceled and then a new file is created from the specified text. If there is no text in the range of line numbers specified or no text in memory at all, no action is taken.

The starting and ending line numbers specified in the OVLSAV statement or the starting and ending line numbers for the entire program (if no line numbers are specified) are saved in the file with the translated text by OVLSAV. Later, when this overlay file is brought into memory by OVLOAD, any existing lines within this range of line numbers are deleted from the program in memory before the fast overlay file is loaded.

Since the optional line numbers in the OVLSAV command are expressions, they are **not** altered by the RENUM command.

NOTE

A fast-overlay file created by the standard memory version of OVLSAV cannot be brought into memory by the extended memory (XM) version of OVLOAD, and vice versa.

Using the Syntax Options:

The named device must be file-structured. If no device is given, the system device is assumed. If the named device does not use the system device driver, its driver must be LOADed into memory before OVLSAV executes. When the device is a serial-access, file-structured device, no files may be OPEN on it. If the drive number is omitted, zero is assumed.

The file name is required. If no file name extension is specified, a default extension of ".BOL" is used.

If the target device is directory-structured (e.g., DK, DL, DY, or DX), the **INTO** option can be used. The (rounded) expression following the keyword INTO stipulates the maximum number of blocks required by the file. When the INTO option is used, the first sufficient empty space on the target device is selected for the file. When the INTO option is not used, one half of the largest empty space on the target device is set aside for the file. In either case if the specified or default space exceeds the actual number of blocks needed for the file, the unused blocks are returned to an empty status. (The INTO option is not supported by the OVLSAV VØ2-Ø1.)

When storing a fast-overlay file on a nearly full disk, use the INTO option. Half the remaining free space may not be large enough for the file. In order to use all the available disk space, you will need to specify the required number of blocks rather than use the default.

The optional expressions are rounded to integers and used as line numbers to delimit the range of the text in memory to be included in the fast overlay file. The expressions must evaluate to numbers between 1 and 32767, inclusive. If only one line number is used, only that line is saved. If two line numbers are used, all program lines between and including those lines are saved. When both line numbers are omitted, all the program text in memory is saved in the file.

PRINT (Nonresident)

Examples:

```
15Ø PRINT A,B,C$
16Ø PRINT #L+2,A+B+C;" IS THE ANSWER"
18Ø PRINT TAB(4Ø);"SUM IS";D
19Ø PRINT 1ØØ,(A*C+(D*D)),3.45EØ5;
2ØØ PRINT
21Ø PRINT,,,,DA$
22Ø PRINT #N,
```

Syntax Form:

```
                                   ┌expression        ┐┌   ┌expression        ┐┐
                                   │array expression  ││   │array expression  ││
[line no.] PRINT [#expression,]    │waveform expression││{,}│waveform expression││...
                                   │string expression ││{;}│string expression ││
                                   │string array      ││   │string array      ││
                                   └TAB(expression)   ┘└   └TAB(expression)   ┘┘
```

Descriptive Form:

```
                                   ┌expression          ┐┌   ┌expression          ┐┐
                                   │array expression    ││   │array expression    ││
[line no.] PRINT [#target plun,]   │waveform expression ││{,}│waveform expression ││...
                                   │string expression   ││{;}│string expression   ││
                                   │string array        ││   │string array        ││
                                   └TAB(column number)  ┘└   └TAB(column number)  ┘┘
```

Purpose:

To output ASCII data to the terminal, a file, or peripheral device.

Discussion:

The PRINT command outputs constants, strings, variables, string variables, waveforms, arrays, and string arrays to the terminal or any peripheral capable of being written to. The information is sent as ASCII characters.

Formatting Your Output:

The comma and the semicolon. Two delimiters are available for separating output items -- the comma and the semicolon. The comma divides a line of printing into fields, 14 print positions wide. It is like hitting the tab key on a typewriter when the tab-stops have been set at positions 14, 28, 42, ... An item following a single comma is printed in the next free field beyond the present position. (In line 28Ø of the example program below, the number 54321 is printed in the fifth field because ST$ extends into the fourth field.) Using two (or more) commas in a row is like hitting a typewriter's tab key that many times -- one (or more) print fields are skipped before the item after the commas is printed. If you specify n commas, n-1 fields are skipped. (Line 29Ø shows this.)

The semicolon causes data to be printed in a concatenated form with no extra spaces between two items. However, when a positive number is output following a semicolon, a space does appear. This space is in place of a plus sign (+). Printing the STR function of the number, instead of just the number, eliminates this space, as in line 38Ø below.

Either of these delimiters, when used as the last character in a PRINT statement, suppresses the carriage return and line feed normally output by each PRINT statement. The result is that the next PRINT statement starts printing from the position held by the previous PRINT statement -- the PRINT statement ending with a comma or semicolon. Remember, however, that a comma will cause a skip to the beginning of the next field, so, if a comma is used, the next PRINT statement will start printing in the next field. (See lines 4Ø, 2ØØ, and 21Ø below.)

Tabulation. The TAB function causes the data item following it to be printed starting at the column number specified. The columns are numbered from zero. If the value of the TAB function is less than the current column position or greater than 2ØØ, the function is ignored. To be effective, the TAB function must be followed by a semicolon. If a comma is used instead of a semicolon, the data item that follows it is printed in the next available field (Lines 41Ø and 42Ø show this).

Blank lines. When a PRINT statement is used with no list of output items, only the carriage return and line feed are output. This normally prints a blank line. An exception to this is if the previous PRINT statement ended with a comma or semicolon. In that case, the second PRINT statement's carriage return and line feed finishes the line started by the previous PRINT statement and no blank line is printed. (In line 15Ø below, the first PRINT completes the previous line, but the next two PRINTs produce two blank lines.)

Array output. When arrays are printed, each element is right justified in a field 14 characters wide, five fields per line. A positive number is preceded by a space; a negative number, by a minus sign. Array output always begins on a new line.

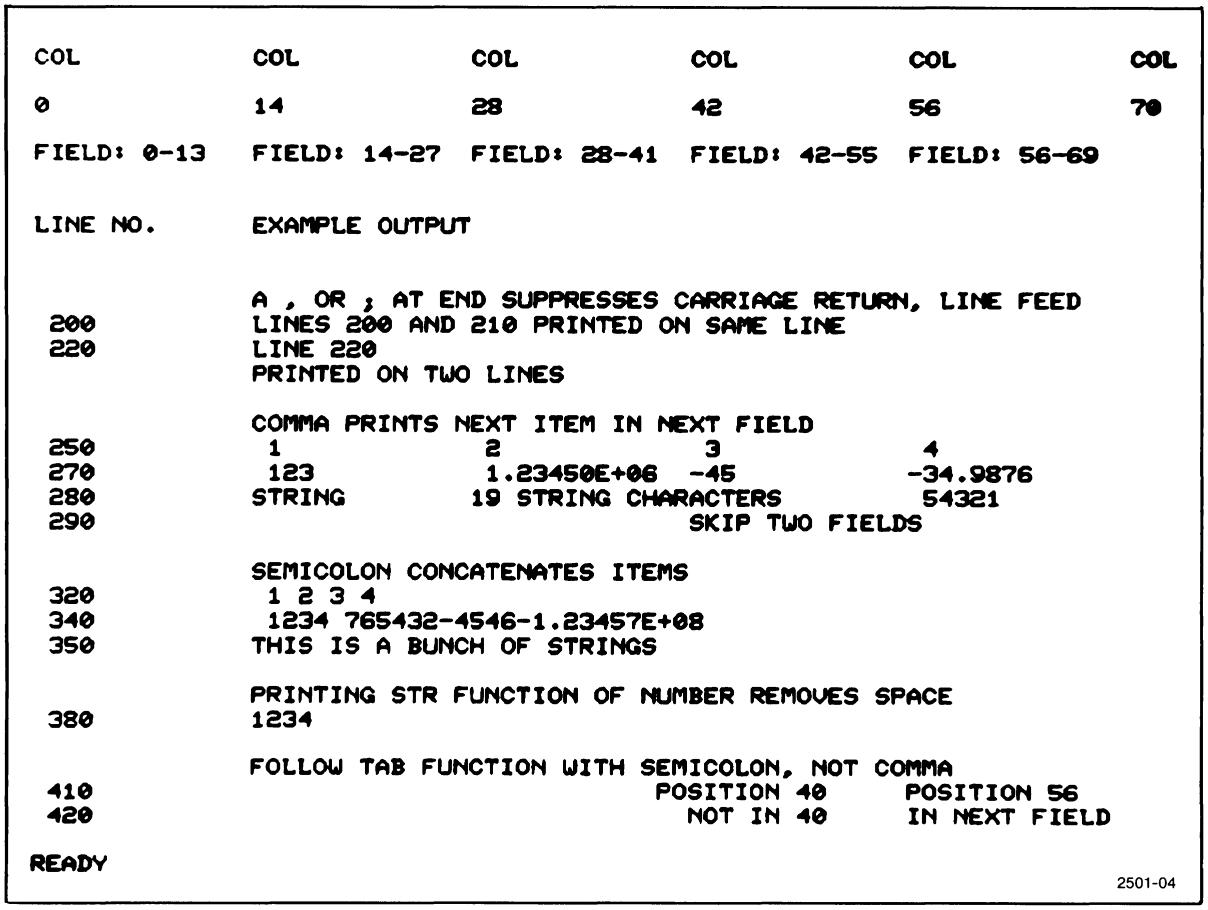
Waveform output. Waveforms are output as four separate data sets. First, the array is printed using the array output format. Next, the data sampling interval, the horizontal units, and finally the vertical units are output, each on a separate line.

String array output. The ability to PRINT a string array has been included to simplify the handling of data from an IEEE 488 Interface. Each element of the string array is printed separately, starting in column zero and is followed by a carriage return and a line feed.

Formatting Example. Here is a sample program that demonstrates some formatting techniques using PRINT statements. The column numbers at the top of the sample program's output (Fig. 4-1) represent the fields in the output line produced by the comma. The first number in each output line corresponds to the line number of the PRINT statement that produced the line.

```
10 REM
20 REM PRINT "COL" AT START OF EACH FIELD
30 REM
40 FOR I=1 TO 6\PRINT "COL",\NEXT I
50 PRINT
60 REM
70 REM PRINT STARTING COLUMN NUMBER FOR EACH FIELD
80 REM
```

```
9Ø FOR I=Ø TO 7Ø STEP 14\PRINT STR(I),\NEXT I
1ØØ PRINT
11Ø REM
12Ø REM SHOW WIDTH OF EACH FIELD
13Ø REM
14Ø FOR I=Ø TO 56 STEP 14\PRINT "FIELD: ";STR(I);"-";STR(I+13),\NEXT I
15Ø PRINT\PRINT\PRINT
16Ø PRINT "LINE NO.","EXAMPLE OUTPUT"
17Ø PRINT\PRINT
18Ø REM
19Ø PRINT,"A , OR ; AT END SUPPRESSES CARRIAGE RETURN, LINE FEED
2ØØ GETLIN X\PRINT X,"LINES";X;" AND";
21Ø GETLIN X\PRINT X;" PRINTED ON SAME LINE"
22Ø GETLINE X\PRINT X,"LINE";X\PRINT,"PRINTED ON TWO LINES"
23Ø PRINT
24Ø PRINT,"COMMA PRINTS NEXT ITEM IN NEXT FIELD"
25Ø GETLINE X\PRINT X,\FOR I=1 TO 4\PRINT I,\NEXT I
26Ø PRINT
27Ø GETLIN X\PRINT X,123,12345ØØ,-45,-34.9876
28Ø ST$="19 STRING CHARACTERS"\GETLIN X\PRINT X,"STRING",ST$,54321
29Ø GETLIN X\PRINT X,,,"SKIP TWO FIELDS"
3ØØ PRINT
31Ø PRINT,"SEMICOLON CONCATENATES ITEMS"
32Ø GETLIN X\PRINT X,\FOR I=1 TO 4\PRINT I;\NEXT I
33Ø PRINT
34Ø GETLINE X\PRINT X,1234;765432;-4546;-1.234567E+08
35Ø GETLIN X\PRINT X,"THIS IS ";"A BUN";"CH OF STRINGS"
36Ø PRINT
37Ø PRINT,"PRINTING STR FUNCTION OF NUMBER REMOVES SPACE"
38Ø GETLIN X\PRINT X,\FOR I=1 TO 4\PRINT STR(I);\NEXT I
39Ø PRINT\PRINT
4ØØ PRINT,"FOLLOW TAB FUNCTION WITH SEMICOLON, NOT COMMA"
41Ø GETLIN X\PRINT X,TAB(4Ø);"POSITION 4Ø","POSITION 56"
42Ø GETLIN X\PRINT X,TAB(4Ø),"NOT IN 4Ø","IN NEXT FIELD"
43Ø END
```



```
COL            COL            COL            COL            COL            COL

0              14             28             42             56             70

FIELD: 0-13    FIELD: 14-27   FIELD: 28-41   FIELD: 42-55   FIELD: 56-69


LINE NO.       EXAMPLE OUTPUT


               A , OR ; AT END SUPPRESSES CARRIAGE RETURN, LINE FEED
 200           LINES 200 AND 210 PRINTED ON SAME LINE
 220           LINE 220
               PRINTED ON TWO LINES

               COMMA PRINTS NEXT ITEM IN NEXT FIELD
 250           1             2             3             4
 270            123          1.23450E+06  -45           -34.9876
 280           STRING        19 STRING CHARACTERS        54321
 290                                       SKIP TWO FIELDS

               SEMICOLON CONCATENATES ITEMS
 320            1 2 3 4
 340            1234 765432-4546-1.23457E+08
 350           THIS IS A BUNCH OF STRINGS

               PRINTING STR FUNCTION OF NUMBER REMOVES SPACE
 380           1234

               FOLLOW TAB FUNCTION WITH SEMICOLON, NOT COMMA
 410                                   POSITION 40     POSITION 56
 420                                     NOT IN 40     IN NEXT FIELD

READY
```

2501-04

Fig. 4-1. Output of example program demonstrating the PRINT command.

PRINTing to a File:

When PRINTing to a file, care must be taken so that the data can be INPUTed correctly. The following statement will provide a file that can be displayed by the COPY command, but cannot be INPUT:

```
PRINT #N,A,B
```

This is because the ASCII characters for A and B will be separated in the file by spaces, which the INPUT command discards. The INPUT command requires a carriage return as a delimiter between data items. (A comma is also a valid delimiter between data items if the first item is numeric.)

The correct way to PRINT multiple data items is as follows:

```
        PRINT #N,A\PRINT #N,B
```

This will place a carriage return between A and B.

When PRINTing arrays, the same care must be taken. Arrays should be PRINTed to a file using a FOR/NEXT loop.

```
        1Ø FOR I=Ø TO SIZ(A)-1
        2Ø PRINT #N,A(I)
        3Ø NEXT I
```

Waveforms should be PRINTed to a file by first PRINTing the array using a FOR/NEXT loop. Then the data sampling interval, horizontal units, and vertical units should be PRINTed with separate PRINT statements to allow the waveform to be INPUT back to memory.

Using the Syntax Options:

When the expression following the pound sign (#) is supplied, it is rounded to an integer and used as the target peripheral logical unit number (PLUN). The PLUN must be between Ø and n, where n is the number of PLUNs allowed at initialization time (default of six).

When a PLUN is omitted, output goes to the terminal (PLUN zero). If a nonzero PLUN is used, it must be OPEN FOR WRITE. If it is OPEN FOR READ or UPDATE, a fatal error results.

Notice that a PLUN of zero is allowed. If you use a variable for the PLUN, a program's output can be directed to one of several peripherals -- the terminal, a line printer, or a file. To direct output to peripherals other than the terminal, simply OPEN the desired device or file FOR WRITE and set the PLUN variable in the PRINT statement to the logical unit number assigned to the file or device by the OPEN statement. Setting the PLUN variable to zero sends output to the terminal.

If the PLUN is given, a comma must follow the PLUN, but this comma does not serve as a formatting delimiter.

Numeric expressions, array expressions, waveform expressions, and string expressions are permitted. However, string arrays may be neither used in expressions nor zoned.

The keyword **TAB** specifies the tabulation printing function explained above. The expression in parentheses is rounded to an integer and used as the column value at which to start PRINTing the next data item. Columns are numbered from zero.

Application Example:

If you PRINT to a file in the same manner you do to a terminal or line printer, you will not be able to INPUT the data from the file back into memory. To be able to INPUT a PRINTed file, each item in the file must be separated by a delimiter acceptable to the INPUT command: a carriage return (or a comma if the items are numeric).

This routine shows some methods for inserting the required delimiters. To separate the numbers in line 3Ø, commas between the numbers are explicitly output to the file. By PRINTing the array in a loop (lines 5Ø to 8Ø), a carriage return separates each array element as it is output. Also each string is printed with a separate statement to delimit it by a carriage return (lines 9Ø and 1ØØ).

```
1Ø  OPEN #1 AS DK1:"TEST.DAT" FOR WRITE
2Ø  PRINT #1,"THIS IS A PRINTED FILE"
3Ø  PRINT #1,10,",",2Ø,",",3Ø,",",4Ø
4Ø  DIM B(3)
5Ø  FOR I=Ø TO 2
6Ø  B(I)=I*1.44
7Ø  PRINT #1,B(I)
8Ø  NEXT I
9Ø  PRINT #1,"THIS IS ONE STRING"
1ØØ PRINT #1,"THIS IS A SECOND STRING"
11Ø CLOSE #1
12Ø REM
13Ø OPEN #1 AS DK1:"TEST.DAT" FOR READ
14Ø INPUT #1 A$,A,B,C,D
15Ø DIM BB(3)
16Ø INPUT #1,BB
17Ø INPUT #1,B$,C$
18Ø CLOSE #1
19Ø RETURN
```

Because explicit delimiters are output, after the file is CLOSEd and OPENed FOR READ, the data can be INPUT into variables (lines 14Ø to 17Ø). If line 3Ø changed to:

```
30 PRINT #1,10,20,30,40
```

and line 7Ø is replaced by:

```
85 PRINT #1,B
```

you would get a data error if you tried to INPUT the file as in lines 13Ø to 18Ø. But you could display the contents of the file by COPYing it to the terminal (KB) or a line printer (LP).

PRIORITY (Nonresident)

Examples:

```
150 PRIORITY 75
160 PRIORITY X+50
```

Syntax Form:

[line no.] **PRIORI** expression

Descriptive Form:

[line no.] **PRIORI**TY level

Purpose:

To change the priority of a program while it is running.

Discussion:

Routines execute in the order of their priority numbers. In the Scheduler queue, a routine with a higher priority number executes before one with a lower number. Also, a routine of higher priority can interrupt the execution of a routine of lower priority. (The Scheduler's priority-base execution process is discussed in Section 1.)

A BASIC program RUNs at a default priority value of 50. With this command you can assign one of 127 priority levels to a running program, 0 as the lowest and 126 as the highest.

A program maintains the priority of the last PRIORITY statement (or the default value of 50) while it is running. If the priority of the system is changed by a PRIORITY statement in a subroutine, the original priority (the priority before the transfer caused by the GOSUB) is restored on RETURN from that subroutine.

The system's priority can also be altered by WHEN or SCHEDULE command interrupt routines (those subroutines that receive program control after an instrument event has occurred or a user-designated time interval has elapsed). The system takes on the priority specified by the associated WHEN or SCHEDULE command. After completion of the interrupt routine, the RETURN command restores the original system priority before returning program control to the main program (or idle mode, depending on when the interrupt occurred).

This priority number is not related in any way to the hardware priorit levels in the processor.

Using the Command Syntax:

The priority level expression is rounded to an integer. It must evaluate to a number between Ø and 126, inclusive.

Uses:

Interrupt routines can be effectively blocked (prevented from executing) by raising the system priority to 126. Likewise, a scheduled routine that has not interrupted the system because of a low priority can be forced to execute by lowering the system priority to zero.

PUT (Nonresident)

Examples:

```
120 PUT "STO" INTO #N,B$
130 PUT A$,X5 INTO #J,X$,Y$
140 PUT "SET?" INTO @Ø,LA,SA
270 PUT A$;A INTO #3;1
```

Syntax Form:

```
                  (expression           )[   (expression           )]
                  |array expression     |[   |array expression     |]
[line no.] PUT   <|waveform expression  >[{,}<|waveform expression  >]... [;]
                  |string expression    |[{;}|string expression    |]
                  (string array         )[   (string array         )]
                     (#expression[;expression][,string expression] ...)
                     |              {array expression        }         |
               INTO <|@expression,  {expression[,expression] }         >
                     |            [ {array expression        } ]       |
                     (            [;{expression[,expression] } ]...    )
```

Descriptive Form:

```
                  (source expression           )[   (source expression           )]
                  |source array expression     |[   |source array expression     |]
[line no.] PUT   <|source waveform expression  >[{,}<|source waveform expression  >]... [;]
                  |source string expression    |[{;}|source string expression    |]
                  (source string array         )[   (source string array         )]
                  (#target ilun [;secondary address]                                     )
                  |             [,driver-dependent specification of data                  |
                  |             or status information to be sent to instrument] ...       |
            INTO <| @IEEE 488 interface number,                                           >
                  |       {listen and secondary address pairs      }                      |
                  |       {listen address [,secondary address]     }                      |
                  |     [ {listen and secondary address pairs      } ]                    |
                  (     [;{listen address [,secondary address]     } ] ...                )
```

Purpose:

To send data to a specified instrument.

Discussion:

This command allows you to control an instrument by setting internal status in the instrument. With some instruments, data in the form of arrays and waveforms can be sent to the device for viewing.

Like the GET command, the PUT command is divided into two parts, the source and the target. The source is the data that is sent to the instrument. This data may be in many forms, depending on the instrument referenced. The target instrument is indicated by either the instrument logical unit number (ILUN) of the ATTACHed instrument or the IEEE 488 interface number followed by one or more addresses. With the ILUN, one or more driver-dependent strings may be used. These strings tell the instrument driver what to do with the source data. Each instrument driver recognizes a different set of strings. For any instrument, only those strings that its driver responds to should be used. Complete documentation of the driver-dependent strings recognized by a driver is found in the manual for that driver.

When the PUT command executes, the instrument must be on line and the required instrument driver must be LOADed in memory. Also either the instrument must be ATTACHed to associate it with the instrument logical unit number (ILUN), or the communication must be through the low-level IEEE 488 Interface driver, GPI.SPS, which is discussed in Section 6.

Using the Syntax Options:

No instrument driver uses all the legal syntax variations of the PUT command. The manual for each driver shows which of the forms are allowed by that driver.

The list of one or more source expressions may include numeric expressions, array expressions, waveform expressions, string expressions, and/or string arrays depending on what a particular driver allows. Multiple source arguments are usually separated by commas, but the syntax of PUT also accepts a semicolon. The semicolon option is used by the high-level

IEEE 488 Instrument driver (GPI.SPS) to suppress the sending of a delimiter (the ASCII code for a comma) between the multiple source items. (INS.SPS is not supported by TEK SPS BASIC VØ2-Ø1.)

The specification following the keyword **INTO** designates the target instrument. If a pound sign (#) is used, the expression after it is the instrument logical unit number (ILUN) of the attached instrument. The optional semicolon and expression is used by the high-level IEEE 488 Instrument driver, INS.SPS, to specify the secondary address of the target IEEE 488 instrument. (INS.SPS is not supported by TEK SPS BASIC VØ2-Ø1.) The optional string expressions are the driver-dependent strings which determine how the data is used.

If an at sign (@) is used, the expression following it is the number of the IEEE 488 interface through which the data is sent. When the at sign is specified, the low-level IEEE 488 Interface driver, GPI.SPS, must be used. The list of expressions after the interface number specifies the primary listen and optional secondary addresses of devices connected to the IEEE 488 interface bus. See Section 6 for complete documentation.

PUTBLK (Nonresident)

Examples:

```
6ØØ PUTBLK DK1:"TEST.DAT",3,B(Ø:255)
15Ø PUTBLK X,A$
49Ø PUTBLK DX:J*2,A+.5
```

Syntax Form:

```
[line no.] PUTBLK [device name[constant]:][string expression,]
                  expression, { string expression }
                               { array expression  }
```

Descriptive Form:

```
[line no.] PUTBLK [device name[drive number]:][file name,]
                  target block number, { source string expression }
                                        { source array expression  }
```

Purpose:

To store a block of data on a directory-structured peripheral device.

Discussion:

The PUTBLK command stores data in a specified block on a directory-structured device. (One block holds 256 16-bit words of data; one word can hold one 16-bit integer or two 8-bit ASCII characters.) The block number in which the data is stored can be given as an absolute block number of the device or as a block relative to the start of a file. The data stored can be either a 256-element array or the first 512 characters in a string.

CAUTION

Careless use of PUTBLK could overwrite the contents of the wrong block of information on the device. With this command, it is possible to corrupt any file on the device, including your copy of TEK SPS BASIC.

Using the Syntax Options:

The device must be directory-structured. If no device is named, the system device is used. If the named device does not use the system device driver, its driver must be LOADed before PUTBLK executes. If the drive number is omitted, zero is assumed.

The expression and the presence or absence of a file name determine in which block on the device the data is stored. The expression, which must result in a non-negative number, is rounded to an integer. If the file name is omitted, that integer is used as an absolute block number of the device, and the data is stored in that block. If a file name is given, that integer is added to the file's starting block number to produce the number of the block where the data is stored. In both cases, the resulting block number must be between Ø and the largest block number of the device, inclusive.

An array or a string can be stored in the specified block. If an array is given, it must be dimensioned or zoned to 256 elements. When the array is floating-point, its elements are truncated to integers when stored in the block. When a string is given, its first 512 characters are stored. If the string has fewer than 512 characters, the remainder of the block is filled with nulls.

PUTLOC (Nonresident)

Examples:

```
2ØØ PUTLOC "1765Ø6","232"
21Ø PUTLOC P1+8,A
```

Syntax Form:

$$\text{[line no.] }\textbf{PUTLOC}\left\{\begin{array}{l}\text{expression}\\ \text{string expression}\end{array}\right\},\left\{\begin{array}{l}\text{expression}\\ \text{string epression}\end{array}\right\}$$

Descriptive Form:

$$\text{[line no.] }\textbf{PUTLOC}\left\{\begin{array}{l}\text{decimal address}\\ \text{octal address}\end{array}\right\},\left\{\begin{array}{l}\text{decimal value to be stored at address}\\ \text{octal value to be stored at address}\end{array}\right\}$$

Purpose:

To deposit a value in a valid controller memory location or in a device address.

Discussion:

The PUTLOC command is used by the PATCH files. This command is not intended for general use.

The PUTLOC command allows a BASIC user to assign a value to a word of controller memory or to a valid interface address (explained below). Only even addresses are acceptable to PUTLOC since only a full word (16 bits) can be referenced by this command.

Valid Addresses:

Standard Memory Systems. One word (16 bits) can produce 2^{16} unique addresses -- Ø to 177777 octal. With byte addressing, this means one of 64K distinct bytes (32K words) can be referenced with a 16-bit address. However, the highest 4K possible word addresses are reserved as peripheral address space for device and register addresses, allowing a maximum of 28K words as controller memory space. Thus for PUTLOC, the valid addresses are **the even (word) controller memory locations (Ø to 157776 octal) plus those reserved addresses to which interfaces are strapped.**

Extended Memory Systems. Systems with memory-management hardware and TEK SPS BASIC VØ2XM software have 18-bit addresses. This permits 256K byte (128K word) addresses of which the highest 4K word addresses are reserved as peripheral address space. (This means, for example, that a peripheral status register which is addressed as 164100_8 in a standard memory system must be addressed as 764100_8 in an XM system.) For extended memory (XM) systems, the valid addresses include the possible even (word) controller memory locations (Ø to 757776 octal) plus the addresses in the peripheral address space to which interfaces are strapped.

If the address given is odd or not valid for your system or controller, a fatal error is issued.

CAUTION

It is possible to corrupt Resident BASIC with this command, forcing a complete software reload.

Using the Syntax Options:

The first argument is the address to which the value is assigned. If the argument is a string, it must be the desired octal address. A string expression should evaluate to a string of no more than eight octal digits. However, in standard memory systems only the lower 16 binary digits are used as the address. In extended memory (XM) systems the lower 18 bits are used. If the argument is numeric, it must be the decimal equivalent to the desired address. A numeric expression is converted to binary and, if necessary, truncated to a 24-bit binary integer. Again, in standard systems only the lower 16 bits are used as the address; while in extended memory systems, the lower 18 bits are used. In any case the resulting address must be even, else an error results.

The second argument is the assigned value. If it is a string expression, it should evaluate to a string of no more than eight octal digits. However, only the lower 16 binary digits are stored in the specified location. If the argument is an expression, the result is truncated to an integer before being assigned to the specified address.

RANDOM (Nonresident)

Example:

```
155 RANDOM X,Y
```

Syntax Form:

[line no.] **RANDOM** floating-point variable,floating-point variable

Descriptive Form:

[line no.] **RANDOM** high-order part of seed, low-order part of seed

Purpose:

To set the seed value of the random number generator or to obtain the current seed value.

Discussion:

The seed is the value used by the random number generator to calculate the next number of the random number sequence. The RANDOM command uses two variables as the high- and low-order parts of a 32-bit integer seed. The first variable sets the high-order 16 bits; the second, the low-order 16 bits. The RND function of TEK SPS BASIC (not to be confused with the RANDOM command) uses the seed value to produce a new psuedo-random number each time the function is called. The actual sequence length is about 2^{31} different numbers. (It is not necessary, however, to execute the RANDOM command in order to use the RND function.)

The RANDOM command can also be used to return the value of the seed.

Each time TEK SPS BASIC is loaded, the seed is initialized to $2^{16}+3$. (This corresponds to the first and second variables being set to 1 and 3, respectively, before RANDOM executes.) The initialization process occurs only at system software load time, not each time the RANDOM command is brought into memory. A statement such as:

```
X=1\Y=3\RANDOM X,Y
```

resets the seed to its value at system load time.

Using the Syntax Options:

To allocate the seed value, each variable should be set to an integer value that is greater than or equal to -2^{15} but less than or equal to $2^{15}-1$. **One variable must contain a nonzero value.** Should a non-integer value be used when setting the seed, it will be truncated to an integer. An out-of-range value is set to the nearest in-range integer.

To return the value of the seed, use the RANDOM command with both variables set to zero. After the command has executed, the two variables contain the seed value, the high-order part in the first variable and the low-order part in the second.

NOTE

The RANDOM command arguments must be simple numeric variables or singly subscripted array elements, not constants or expressions.

Uses:

The RANDOM command can be used to start the random number generator at a predetermined point. This allows a program to produce the same random numbers each time it is run. This is a very convenient tool for debugging programs or repeating statistical analyses.

Also, continuous, non-overlapping random sequences can be generated from separate program runs by using the RANDOM command. This is done by returning the current seed at the end of each run and then setting the seed variable to those values prior to the next run.

READ

Examples:

```
1ØØ READ #A,A(1Ø:3Ø),C$
15Ø READ #3,A1,C(3)
```

Syntax Form:

$$\text{[line no.] } \textbf{READ } \#\text{expression,} \left\{\begin{array}{l}\text{variable}\\\text{array}\\\text{waveform}\\\text{string variable}\end{array}\right\} \left[, \left\{\begin{array}{l}\text{variable}\\\text{array}\\\text{waveform}\\\text{string variable}\end{array}\right\} \right] \ldots$$

Descriptive Form:

$$\text{[line no.] } \textbf{READ } \#\text{source plun,} \left\{\begin{array}{l}\text{target variable}\\\text{target array}\\\text{target waveform}\\\text{target string variable}\end{array}\right\} \left[, \left\{\begin{array}{l}\text{target variable}\\\text{target array}\\\text{target waveform}\\\text{target string variable}\end{array}\right\} \right] \ldots$$

Purpose:

To input data from a peripheral device or a data file filled by the WRITE command, allowing floating-point, integer, or ASCII values to be assigned to the specified variables.

Discussion:

Although INPUT and READU also input data to a program, READ is the most commonly used command to input data stored on a peripheral device. It reads data files filled by the WRITE command or by the GET command when data-logging (sending data directly to a peripheral from an instrument). READ brings in the data stored on a peripheral, and assigns values to variables, arrays, waveforms, and string variables.

The READ command accesses a file or device by its peripheral logical unit number (PLUN), not by name. Thus, before a file or device can be read, it must be OPENed FOR READ in order to assign a PLUN to it. If the peripheral is OPEN FOR WRITE or UPDATE, a fatal error results.

The file or device is read in a sequential manner starting at the beginning of the file with the first READ statement. Subsequent READs to the same file continue reading data from where the previous READ ended. (By using a RESET statement you can begin READing from the first of the file again without closing and reopening the file.)

The variables listed in the READ statement must match the data types available to read. For example, if you want to define C$, the next piece of information in the specified file must be a string. (The maximum length of a string that can be READ is 388 characters.) Likewise, floating-point and integer variables specified in the READ statement must match the data coming from the peripheral. For this reason you need to know the order in which various data types were output to the peripheral.

Array dimensions, however, need not be consistent. Arrays can be read either as entire arrays, as a series of smaller subarrays, or as individual elements. You can read the array as two or more smaller arrays by specifying in the READ statement two or more arrays of smaller dimension than the source array. You can read an array as individual elements by specifying array elements, perhaps with a FOR/NEXT loop. (If you specify a simple numeric variable, it will be auto-dimensioned as explained later.)

[The READ command can tell the type of data in a file by the data descriptors that are written into the file along with the data by the WRITE command (or by GET when data logging). The data descriptors delimit the data in the file and inform the READ command of the size and type of the data that is next in the file. These data descriptors are not something a BASIC user needs to be concerned with unless the file being output is to be accessed by software other than TEK SPS BASIC. (The TEK SPS BASIC data descriptors are described in Appendix E.) Because the WRITE command writes data descriptors on a file and stores numbers as well as ASCII characters, a file output by WRITE and input by READ is sometimes called a formatted binary file.]

If the size of the data file is unknown, the EOF statement may be used to cause program control to transfer to a specified line when the end of the file is reached.

If a Control-P is typed at the keyboard while a waveform or array is being input, the entire array (or subarray) is read before program execution terminates.

Auto-dimensioning:

If a simple numeric variable is specified in the READ statement and the next piece of data to be read is an array or the remainder of a partially read array, the simple numeric variable is auto-dimensioned to the size of the array (or the remainder of the array). Auto-dimensioning may cause auto-loaded, nonresident commands to be released if room is needed for the array.

Using the Syntax Options:

The expression following the pound sign (#) is the peripheral logical unit number (PLUN) from which the data is read. The expression, when evaluated and rounded to an integer, must be between 1 and n, where n is the number of PLUNs allowed at initialization time (default of six). The terminal keyboard, PLUN zero, may not be specified.

The list of variables to be assigned values by the READ statement may include integer or floating-point variables, integer or floating-point arrays, waveforms, and string variables. String arrays are not allowed. The type of the variable must match the type of the next value in the file.

Application Example:

READ is often used to retrieve waveforms or arrays stored in a disk file. For example, the following routine reads in an unknown number of waveforms from a file.

```
100 WAVEFORM WA IS A(511),DS,H$,V$
110 OPEN #1 AS DX1:"DATA.FIL" FOR READ
120 EOF #1 GOTO 800
130 READ #1,WA
140 REM
150 REM ROUTINE TO PROCESS EACH
160 REM WAVEFORM GOES HERE
170 REM
 .
 .
 .
790 GOTO 130
800 CLOSE #1
810 RETURN
```

The desired file is assigned PLUN 1 by an OPEN statement (line 11Ø). Then from a loop (lines 13Ø to 79Ø), each waveform is individually read in and processed before the next is read. When the file is empty, the EOF statement (line 12Ø) transfers program control out of the loop to where the file is CLOSEd (line 8ØØ) before the routine terminates.

READU (Nonresident)

Examples:

```
150 READU #1,A,B
160 READU #N,C,DA,D$=SF,B
170 READU #2<9>,A,A$=10
180 READU #J<K>,X,Y,Z,T$=L,A
```

Syntax Form:

$$\text{[line no.] } \mathbf{READU}\ \#\text{expression[<expression>]}, \left\{\begin{array}{l}\text{variable}\\ \text{array}\\ \text{string variable = expression}\end{array}\right\}$$

$$\left[, \left\{\begin{array}{l}\text{variable}\\ \text{array}\\ \text{string variable = expression}\end{array}\right\}\right]\ldots$$

Descriptive Form:

$$\text{[line no.] } \mathbf{READU}\ \#\text{source plun [<record number>]},$$

$$\left\{\begin{array}{l}\text{target variable}\\ \text{target array}\\ \text{target string variable = number of characters in string}\end{array}\right\}$$

$$\left[, \left\{\begin{array}{l}\text{target variable}\\ \text{target array}\\ \text{target string variable = number of characters in string}\end{array}\right\}\right]\ldots$$

Purpose:

To read DEC RT-11 FORTRAN-compatible data files (files written without TEK SPS BASIC data descriptors) and record I/O files (TEK SPS BASIC random-access files).

Discussion:

Data files that have been created by a DEC RT-11 FORTRAN program or by the WRITEU command can be read with this command. The variable names specified in the READU command determine how much information is transferred from the file to the program. Waveforms may not be specified in this command. READU inputs the data in as many bytes (eight bits per byte) as the data type of the variables requires. The following table describes the amount of data transferred for the four possible variable types:

Floating-point variable	4 bytes
Floating-point array	4 bytes per element
Integer array	2 bytes per element
String variable	number of bytes specified by expression following equal sign (=).

The following example demonstrates how the READU command can be used to input various data types.

```
100 INTEGER I(2)
110 DIM A(2)
 .
 .
 .
500 OPEN #N AS F$ FOR READ
510 READU #N,A,C$=10,I,X
```

The data will be input as follows:

A(Ø)	gets first 4 bytes
A(1)	gets next 4 bytes
A(2)	gets next 4 bytes
C$	gets next 1Ø bytes (string of length 1Ø)
I(Ø)	gets next 2 bytes
I(1)	gets next 2 bytes
I(2)	gets next 2 bytes
X	gets next 4 bytes

Altogether, the READU command in line 51Ø reads in 32 bytes of data.

[The files read by the READU command contain neither the TEK SPS BASIC data descriptors nor delimiters between data items. Thus, READU cannot tell the type or size of the data it is to read. For this reason, the files input by READU are sometimes called unformatted binary files.] Because BASIC has no means of determining the type of data in the file, it is your responsibility to read in the data in the same order as it was written. It is possible, for example, to inadvertently read a string into a floating-point array.

With the regular form of READU, the data is read sequentially, starting at the beginning of a file with the first READU statement. Subsequent inputs from the same file continue reading where the previous READU ended.

When the record I/O (input/output) form of READU is indicated -- by the presence of the angle brackets (<>) -- the mode of access is random. Any data record in the file may be read in any order. The record I/O form of READU determines where in the file the data is read from by multiplying the given record number by the data record length. The length of the data record is computed by summing the byte count of the variables in the input list of the READU statement. (The byte count used for each type is discussed above.) For example, if A is an array of 25 elements, the statement:

```
READU #N<7>,A$=10,A
```

reads in a data record 110 bytes long starting with the 770th byte of data in the file assigned PLUN #N. READU calculates this by first computing the byte count for the given variables (25*4+10) and then multiplying this result of 110 by 7, the given record number.

The data record length is calculated each time a record I/O form of READU is executed. So when using record I/O, you must input or output an entire data record with each READU or WRITEU statement, even if you want to access only a part of a record.

The READU command accesses a file by its peripheral logical unit number (PLUN), not by file name. Before executing READU, the file must be OPEN FOR READ or UPDATE depending on which form of READU is used. When the regular (sequential-access) form of READU is used, the file must be OPEN FOR READ. To use the record I/O (random-access) form of READU, the device must be directory-structured and the file must be OPEN FOR UPDATE.

Using the Syntax Options:

The expression following the pound sign (#) is the peripheral logical unit number (PLUN) from which the data is read. The expression, when evaluated and rounded to an integer, must be between 1 and n, where n is the number of PLUNs allowed at initialization time (default of six). The terminal keyboard, PLUN zero, may not be specified.

The optional expression in angle brackets (<>) specifies the record I/O form of READU. The expression, when evaluated and rounded to an integer, is used as the number of the data record to be read. **The records are numbered from zero.** When the angle brackets and expression are omitted, the regular form of READU is assumed.

The list of variables to be assigned values may include floating-point variables, floating-point or integer arrays, and string variables. String arrays are not allowed. A string variable must be followed by an equal sign (=) and an expression indicating the number of characters (bytes) to assign that string variable.

Application Example:

To illustrate how useful record I/O is, consider this binary search routine. Here we assume that there are two data files: a name file and an information file. In the name file each record consists of only the name (2Ø bytes long) and a pointer -- the record number of the record in the information file where the rest of the data about that person is kept. The first record (record zero) of the name file has a blank name field and contains the number of names in the file in the pointer field. In the information file each record is 1ØØ bytes long. The first 4Ø bytes is the person's street address stored as two ASCII strings, 2Ø characters (2Ø bytes) each.

To get the data stored in the information file for a given person, the name file is searched for the desired name. Then the number stored with the name is used to access the record in the information file where the rest of the data is stored. Using the record I/O form of READU allows random access of the records in both files.

The name file is kept in alphabetical order. Being small, compared with the information file, it is much faster to sort and much easier to keep in alphabetical order as names are added to or deleted from the file. By having the file in alphabetical order and by using record I/O, a binary search can be used to find the desired name. [The binary search keeps dividing the search area in half as it zeros in on the name it is looking for. So, compared to the number of names in the file, it needs few tries to find the desired name. In fact, a binary search of a record I/O file requires at most p reads of the file where p is the nearest power of 2 greater than or equal to the number of records in the file (i.e., 2^p => number of records). So, a binary search needs no more than p tries to find a match.] Depending on the size of the file, a binary search can be considerably faster than a serial search.

The example below is a subroutine that prints a part of a data record from the information file -- the street address. It asks for the desired name (line 53Ø) and then calls the binary search routine to find the record number where the data is stored, R, (line 58Ø). If the name is not in the file, the search routine returns a negative record number. The address subroutine, after checking to be sure a match was found (line 61Ø), reads the entire record pointed to (line 65Ø). It then prints the address which is stored in the first 4Ø bytes of the record (lines 67Ø and 68Ø).

```
5ØØ REM SUBROUTINE TO FIND ADDRESS
51Ø REM
52Ø REM GET SEARCH NAME
53Ø PRINT "WHOSE ADDRESS DO YOU WANT";
54Ø INPUT S$
55Ø REM SEARCH NAME FILE FOR NAME
56Ø REM GET RECORD NUMBER OF DATA
57Ø REM IN INFORMATION FILE
58Ø GOSUB 1ØØØ
59Ø REM IF RECORD NUMBER NEGATIVE, EXIT
6ØØ REM NAME WAS NOT IN FILE
61Ø IF R<Ø THEN RETURN
62Ø REM OPEN INFORMATION FILE FOR UPDATE
63Ø OPEN #2 AS DX1:"INFO.FIL" FOR UPDATE
64Ø REM READ IN RECORD POINTED TO
65Ø READU #2<R>,F$=1ØØ
66Ø REM PRINT ADDRESS FROM RECORD
67Ø PRINT SEG(F$,1,2Ø)
68Ø PRINT SEG(F$,21,4Ø)
69Ø REM CLOSE FILE, RETURN
7ØØ CLOSE #2
```

```
710 RETURN
720 REM
730 REM BINARY SEARCH OF NAME FILE
740 REM
750 REM OPEN NAME FILE FOR UPDATE
1000 OPEN #1 AS DX1:"NAME.FIL" FOR UPDATE
1010 REM READ FIRST RECORD FOR NUMBER OF NAMES
1020 READU #1<0>,NA$=20,N
1030 REM INITIALIZE LOWER,UPPER BOUNDARIES
1040 L=1
1050 U=N
1060 REM IF UPPER < LOWER, EXIT (NO MATCH)
1070 IF U<L THEN 1280
1080 REM FIND THE APPROXIMATE MIDDLE RECORD
1090 REM IN SEARCH INTERVAL
1100 M=ITP((L+U)/2)
1110 REM READ MIDDLE RECORD
1120 READU #1<M>,NA$=20,R
1130 REM IF MATCH, EXIT SEARCH SUBROUTINE
1140 IF S$=TRM(NA$) THEN 1310
1150 REM IF SEARCH NAME < NAME IN MIDDLE RECORD
1160 REM LOOK IN TOP HALF OF SEARCH INTERVAL
1170 IF S$<NA$ THEN 1240
1180 REM IF SEARCH NAME > NAME IN MIDDLE RECORD
1190 REM LOOK IN BOTTOM HALF OF SEARCH INTERVAL
1200 REM ADJUST LOWER BOUNDARY UPWARD TO MIDDLE +1
1210 L=M+1
1220 GOTO 1070
1230 REM ADJUST UPPER BOUNDARY DOWNWARD TO MIDDLE -1
1240 U=M-1
1250 GOTO 1070
1260 REM IF NO MATCH PRINT MESSAGE
1270 REM AND RETURN NEGATIVE RECORD NUMBER
1280 PRINT "NAME NOT FOUND IN FILE"
1290 R=-1
1300 REM CLOSE FILE AND RETURN
1310 CLOSE #1
1320 RETURN
```

Before the binary search can start, the routine needs to know how many names are in the file. Reading the number in the pointer field of the first record (record zero) gives the number of names, N (line 1020). The binary search of the names is done by looking at the approximate middle record, M, of the search interval. If a match is found, you're done. But if not, the search interval is cut in half and the middle record of the new interval (line 11ØØ) is examined for a match. The search continues until a match is found (line 114Ø) or the search interval becomes less than one (line 1Ø7Ø).

The first search interval is the entire file so the lower boundary, L, is set to 1 and the upper boundary, U, to N. As the search zeros in on the name, the interval is made smaller and smaller. Depending on whether the search name is less than or greater than the name in the currently examined record, the new search interval is the top or bottom half of the old search interval. This means, either the upper boundary is decreased to 1 less than the midpoint (line 124Ø), or the lower boundary is increased to 1 more than the midpoint (line 121Ø).

RELEASE

Examples:

```
600 RELEASE ALL
605 RELEASE "FFT","WAIT","CT"
620 RELEASE T$(J),B$
630 RELEASE AUTO
```

Syntax Form:

```
                       ( (string expression) [ (string expression) ]     )
[line no.] RELEASE     { (AUTO             ) [,(AUTO             ) ] ... }
                       ( ALL                                             )
```

Descriptive Form:

```
                       ( (driver or command name   ) [  ( driver or command name   ) ]     )
[line no.] RELEASE     { (all AUTO-loaded commands ) [ ,( all AUTO-loaded commands ) ] ... }
                       ( ALL drivers and nonresident commands in memory                    )
```

Purpose:

To remove peripheral and instrument drivers and nonresident commands from memory.

Discussion:

Modules that have been loaded with the LOAD command, as well as auto-loaded nonresident commands, are released from memory with this command. If a module was loaded with the LOAD command, the RELEASE command is the only way to remove it from memory. (Auto-loaded nonresident commands are released automatically when room is needed for arrays, program text, another nonresident command, or a driver.)

Using the Syntax Options:

The string expression must evaluate to the file name of the command or driver being RELEASEd. An extension need not be specified but if it is, it must be .SPS. (The .SPS extension is reserved for nonresident command and driver file names.)

If the keyword **AUTO** is used, all auto-loaded nonresident commands are deleted.

If the keyword **ALL** is used, all nonresident commands and drivers (except the system device driver, the keyboard driver, and any drivers of ATTACHed instrument or OPENed peripherals) are deleted from memory.

REM

Examples:

```
100 REM  THIS STATEMENT CAN BE USED
102 REM  TO INSERT COMMENTS IN YOUR PROGRAMS
```

Syntax Form:

[line no.] **REM** text

Purpose:

To allow insertion of comments in the body of a program.

Discussion:

No action is taken when a REM statement is encountered in a program. Its purpose is solely to allow textual remarks to be included in a program for documentation.

No commands may follow the REM statement on a line. This is because the statement delimiter (\) is considered to be part of the remark. Execution continues with the next program line.

It should be noted that some processing time (though minimal) is spent in recognizing and jumping past the REM statement. For maximum efficiency, REM statements should not be put inside loops in your program.

Also, although it is permitted in BASIC, program control should not be transferred to a REM statement. That is, a REM statement should not be the designated statement in a GOTO, GOSUB, IF, EOF, ONERR, SCHEDULE, WHEN, or other such command. The reason is that you may want to reduce the size of the program by removing the REM statements with the CHANGE command prior to execution. (See the CHANGE command for a discussion on how to do this.) If you have transferred control to a REM statement, you can't later remove it without rewriting part of your program.

RENAME (Nonresident)

Examples:

```
50 RENAME "THIS" TO "THAT"
60 RENAME DX1: "BASICF.DAT" TO "NEW.FIL"
```

Syntax Form:

[line no.] **RENAME** [device name[constant]:]string expression **TO** string expression

Descriptive Form:

[line no.] **RENAME** [device name[drive number]:]old file name **TO** new file name

Purpose:

To change the name of a file on a directory-structured device.

Discussion:

The first file name specified is the old name. This name must currently exist in the directory of the specified device. The second file name is the new name. This is the name that the file will have after execution of the command. If a file already exists on the specified device with the new file name, an error is issued.

Using the Syntax Options:

The device must be directory-structured. If no device is specified, the system device is assumed. If the specified device does not use the system device driver, its driver must be LOADed into memory before RENAME executes. If the drive number is omitted, zero is assumed.

The first string expression is the file name being changed. The string expression following the keyword **TO** is the file name it will have after the RENAME command executes.

RENUM (Nonresident)

Examples:

```
50 RENUM
60 RENUM STEP 5
   RENUM 100,200 TO 3000 STEP 5
   RENUM 100,300 STEP 20
   RENUM 100,200
   RENUM 500 STEP 20
```

Syntax Form:

[line no.] **RENUM** [expression[,expression]] [**TO** expression] [**STEP** expression]

Descriptive Form:

[line no.] **RENUM** [line number[starting,line number ending]]
[**TO** new line number starting] [**STEP** increment]

Purpose:

To renumber BASIC program line numbers in memory.

Discussion:

This command renumbers all or part of your program. It does not allow you to overlay any existing line numbers, or to rearrange the flow of your program lines. All GOSUB and GOTO statements in memory are updated to point to the new line numbers, even those outside the range of lines being renumbered. This includes statements with an implied GOTO or GOSUB such as WHEN, SCHEDULE, and some UNSCHEDULE, EOF, IF, and ONERR statements.

NOTE

RENUM updates explicit line numbers, not expressions used as line numbers. Thus, RENUM does not alter the optional line number expressions in CHAIN, CHANGE, LIST, OVLSAV, REPLACE, RENUM, or SAVE.

Using the Syntax Options:

All expressions in the command are **truncated** to integers between 1 to 32767.

The first and second optional arguments delimit the part of a program to RENUMber. The first expression is the old starting line number -- where renumbering begins. The second is the old ending line number -- where renumbering ends. All lines between and including these lines are renumbered. When the second expression is omitted, renumbering continues to the end of the program. **Omitting both of the first two expressions renumbers the entire program.**

The expression following the keyword **TO** defines the new starting line number. When this is omitted, what is used as the new starting line number depends on what else is specified or not specified. If an old starting line number is present, it remains as the new starting line number. Otherwise, the step size (default ten) is used as the new starting line number.

The expression following the keyword **STEP** specifies the step (increment) size between renumbered lines. When omitted, the default is ten.

Application Example:

The RENUM command allows you to expand a section of a program by many lines. For example, to allow room to insert more than nine lines of code between line 1ØØ and line 11Ø, type:

```
RENUM 11Ø TO 2ØØ
```

This renumbers all lines from the old line 11Ø onward (at the default step size of ten) to 2ØØ, 21Ø.... Now you could insert up to 99 new lines between lines 1ØØ and 2ØØ.

The same technique can be used to make an overlay section from lines 1ØØØ to 1999 in a main program. If, for instance, the section of code to be overlaid is from 515 to 95Ø, you first make room for more lines beyond line 95Ø by typing:

```
RENUM 951 TO 2ØØØ
```

Then, renumber the overlay section with:

```
RENUM 515,95Ø TO 1ØØØ
```

Next, SAVE or REPLACE the main program. Finally, OLD in the overlay program segment and renumber its starting line number to 1ØØØ with:

```
RENUM TO 1ØØØ
```

Check the overlay program's new ending line number. If its final line number is larger than 1999, choose a smaller step size than the default of ten, or make your overlay area in the main program larger.

REPLACE (Nonresident)

Examples:

```
150 REPLACE "PROG2"
    REPLACE DK1:"SUB",1000,2000
    REPLACE CT:/R,F$
    REPLACE DL2:"MAILST" INTO 10
```

Syntax Form:

[line no.] **REPLAC** [device name[constant]:] $\left[/ \begin{Bmatrix} F \\ R \end{Bmatrix} [,] \right]$ string expression

[**INTO** expression][,expression[,expression]]

Descriptive Form:

[line no.] **REPLACE** [device name[drive number]:] [/forward or reverse switch[,]]
program file name [**INTO** number of blocks]
[,line number [starting, line number ending]]

Purpose:

To allow the user to update a previously SAVEd program.

Discussion:

The REPLACE command cancels the specified file and then saves all or part of the program text that is in memory in a program file of the same name. If the given file does not exist on the device, the program text is simply saved.

Specifying the one or two optional line numbers allows you to save only part of the program text in memory. If there is no text in the range of the line numbers specified (or no text in memory at all) the REPLACE command takes no action; no file is canceled and nothing is output to the device. This situation generates no error.

Since the optional line numbers in the REPLACE command are expressions, they are **not** altered by the RENUM command.

If a line of text is longer than 8Ø characters, a warning error is issued. The line in question should be corrected and the file REPLACEd. The OLD command, used to load programs that have been SAVEd or REPLACEd, does not accept lines longer than 8Ø characters.

If the device is directory-structured, the SQUISH command may be used to compact into one area any noncontiguous free (unused) blocks that are created by the REPLACE and CANCEL commands.

[For a serial tape device, if the old file is located, its name is changed to "*EMPTY". No change is made to the data in the file. The new program file is written after the last file on the tape. Since you cannot SQUISH a serial tape device, no free space is gained on the device.]

For faster execution of segmented programs use the OVLSAV command instead of REPLACE and then use the OVLOAD command instead of OVERLAY.

Using the Syntax Options:

The named peripheral must be a file-structured device. If no device is given, the system device is assumed. If the specified device does not use the system device driver, its driver must be LOADed into memory before REPLACE executes. If the device is a serial tape device, no files may be OPEN on it. If the drive number is not specified, zero is assumed.

[The Forward or Reverse switches **/F** or **/R** may be included in the command if the peripheral is a serial tape device. The switch specifies the direction of the tape movement when searching for a file. If the switch is omitted, the tape is rewound before a forward search for the file is made. Searching stops when the file or an end-of-tape is reached. The /F or /R switch is ignored when the device is not a serial tape device.]

Since the device is file-structured, a file name is required. Carelessly specifying the wrong file name can REPLACE a file that you do not want to alter. If no extension is present in the file name, .BAS is assumed.

If the target device is directory-structured (e.g., DK, DL, DY, or DX), the **INTO** option can be used. The (rounded) expression following the keyword INTO stipulates the maximum number of blocks required by the file being saved. When the INTO option is used, the first sufficient empty space on the target device is selected for the file. When the INTO option is not used, one half of the largest empty space on the target device is set aside for the file. In either case, if the specified or default space exceeds the actual number of blocks needed for the file, the unused blocks are returned to an empty status. (The INTO option is not supported by REPLACE VØ2-Ø1.)

When storing a program file on a nearly full disk, use the INTO option. Half the remaining free space may not be large enough for the file. In order to use all the available disk space, you will need to specify the required number of blocks rather than use the default.

The optional expressions are rounded to integers and used as line numbers. They must evaluate to numbers between 1 and 32767, inclusive. If one line number appears, only that line is saved. When two line numbers are used, all program lines between and including those lines are saved. When the line numbers are omitted, all the program lines in memory are saved.

RESCHEDULE (Nonresident)

Examples:

```
150 RESCHEDULE
290 RESCHEDULE STACK
170 RESCHEDULE WITH 51
340 RESCHEDULE STACK WITH N+5
```

Syntax Form:

[line no.] **RESCHE** [**STACK**] [**WITH** expression]

Descriptive Form:

[line no.] **RESCHE**DULE [**STACK**] [**WITH** priority level]

Purpose:

To put either the current job or the routine at the top of the Scheduler stack back on the Scheduler queue and to allow the Scheduler to select a new current job.

Discussion:

The RESCHEDULE command alters the contents of the Scheduler, which is the mechanism in the BASIC operating system that controls the flow of program execution. (The function and parts of the Scheduler are explained in Section I.) RESCHEDULE removes either the current job or the routine at the top of the Scheduler stack and places it back on the Scheduler queue. RESCHEDULE then performs a RETURN. This means that after the change is made, the Scheduler chooses a new current job from between the routine at the top of the stack and the routine at the top of the queue (the highest priority routine). The top of the stack is the new current job unless the top of the queue has a higher priority.

When the routine is RESCHEDULEd, its priority can be optionally changed. This command does not change the task number associated with the routine.

Using the Syntax Options:

If the optional keyword **STACK** is used, the routine at the top of the stack is removed from the stack and placed back in the queue. Then the currently executing routine (the routine in which the RESCHEDULE command appears) is terminated. When this keyword is omitted, the currently executing routine is suspended and placed back in the queue. The line number associated with this routine is the line number of the command following the RESCHEDULE command. When this routine resumes (by its packet being popped off the queue), execution continues with the command following the RESCHEDULE command.

The optional keyword **WITH** and its expression specify the priority assigned the routine being RESCHEDULEd. The expression when rounded to an integer must be between Ø and 126, inclusive, where 126 has highest priority. If this keyword and expression are omitted, the priority of the routine is unchanged.

Application Example:

Time slicing is a method of dividing the use of a device among several applications. The example below shows how to use RESCHEDULE to produce a form of time slicing in which three tasks of equal priority take turns executing. A fourth, high priority task oversees the turn taking by periodically interrupting the currently executing task. Each time it takes control, this fourth task SCHEDULEs itself to interrupt again and RESCHEDULEs the interrupted task, allowing the next task to have a turn.

```
100 REM TASK Ø
110 REM
120 LOAD "CLK"
130 CLOSE #1
140 OPEN #1 AS LP: FOR WRITE
150 SCHEDULE WITH 1 AS TASK 1 GOSUB 1000
160 SCHEDULE WITH 1 AS TASK 2 GOSUB 2000
170 SCHEDULE WITH 1 AS TASK 3 GOSUB 3000
```

```
180 SCHEDULE AFTER .2 WITH 126 AS TASK 4 GOSUB 4000
190 RETURN
1000 REM
1010 REM TASK 1
1020 REM
1030 PRINT #1,"A";
1040 GOTO 1000
2000 REM
2010 REM TASK 2
2020 REM
2030 PRINT #1,"B";
2040 GOTO 2000
3000 REM
3010 REM TASK 3
3020 REM
3030 PRINT #1,"C";
3040 GOTO 3000
4000 REM
4010 REM TIME SLICING TASK
4020 REM
4030 SCHEDULE AFTER .2 WITH 126 AS TASK 4 GOSUB 4000
5040 PRINT #1,"  ";
5050 RESCHEDULE STACK
```

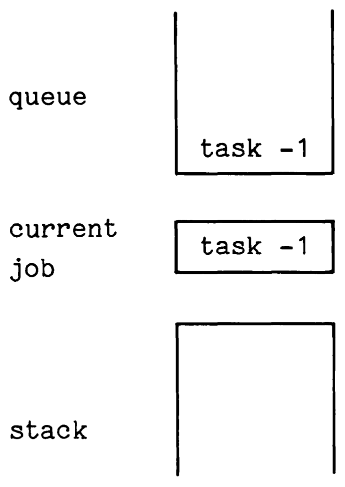
Typing RUN causes line 100 to start executing as task 0. This task loads the clock driver needed by the SCHEDULE command and OPENs the line printer for output. It then puts tasks 1,2, and 3 in the Scheduler queue and task 4 in the clock queue. The RETURN in line 190 signals the end of task 0. Then task 1 is popped off the Scheduler queue and starts executing the loop at line 1000.

Approximately two tenths (0.2) of a second after task 4 (the time slicing task) is put in the clock queue, the clock driver transfers task 4 to the Scheduler queue. Its high priority (126) puts it at the front of the queue. Then, since its priority is higher than task 1's, it interrupts task 1. Task 1 is pushed onto the Scheduler stack and task 4 becomes the current job.

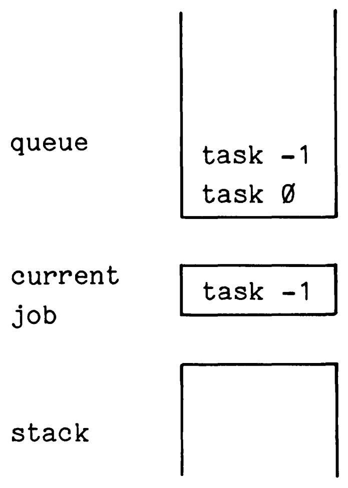
As task 4 executes it SCHEDULEs itself for 0.2 of a second later by putting task 4 (itself) back in the clock queue. Then, just before exiting, it RESCHEDULEs the task it interrupted. That is, it removes task 1 from the stack and places it back on the Scheduler queue. Since tasks 1, 2, and 3 have the same priority, task 1 is put into the queue behind task 3.

With task 1 removed, only the idle packet (task -1) is left on the stack. When task 4 completes, task 2 is popped off the Scheduler queue and starts executing its loop at line 2ØØØ. It continues until its timed turn is over -- until the Ø.2 of a second has elapsed and task 4 is once again put in the front of the Scheduler queue by the clock driver. Then task 4 interrupts task 2 and does what it did before, except this time it takes task 2 from the stack and puts it back on the Scheduler queue (behind task 1). After task 4 finishes, task 3 has its turn as the current job, executing the loop at line 3ØØØ until task 4 interrupts again to give task 1 a second turn, and so on.

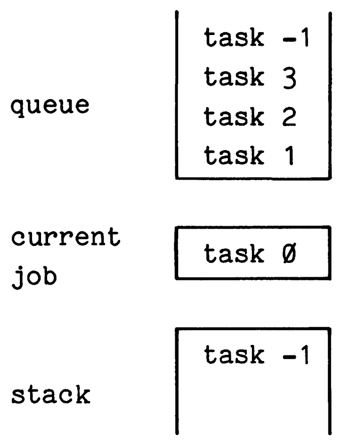
The following set of diagrams illustrates the action of the Scheduler while this time-slicing program runs:



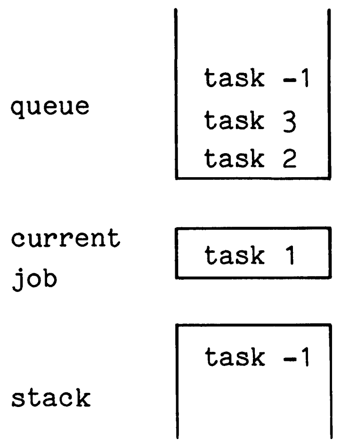
A. In idle mode, BASIC patiently waits with the idle packet (task -1) occupying both the current task and the front of the queue.



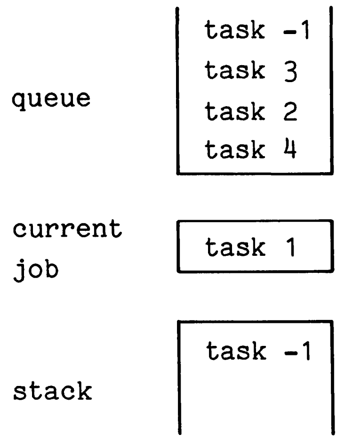
B. Typing RUN enters task Ø into the front of the queue.



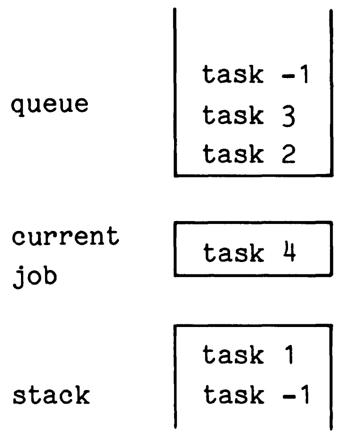
C. Task Ø replaces task -1 which is pushed onto the stack. Task Ø puts tasks 1,2, and 3 into the queue and task 4 into the clock queue.



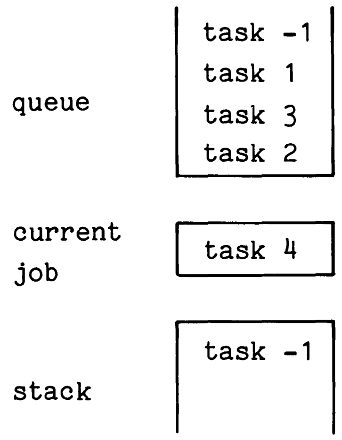
D. When a RETURN is encountered, task Ø completes and task 1 is popped off the queue. Task 1 starts executing its loop.



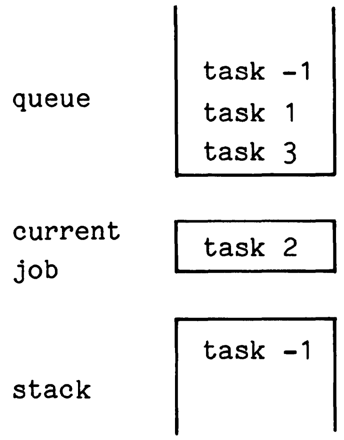
E. After approximately Ø.2 of a second, task 4 is put in the queue by the clock driver. Its high priority puts task 4 at the front of the queue.



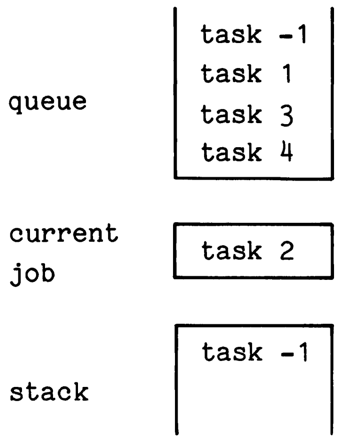
F. Because of its higher priority, task 4 interrupts task 1. Task 1 is pushed onto the stack.



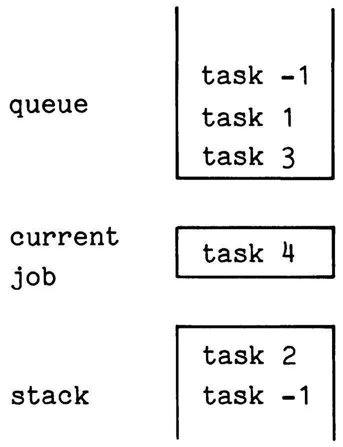
G. Task 4 puts itself into the clock queue and removes task 1 from the stack -- placing it back on the queue.



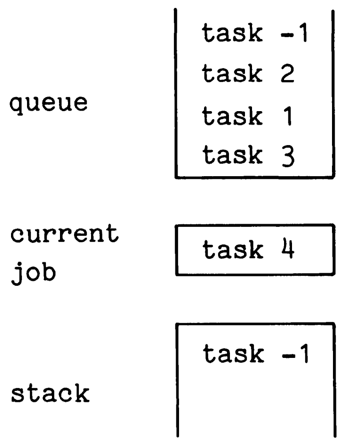
H. When task 4 completes, task 2 is popped off the queue. Task 2 starts executing its loop.



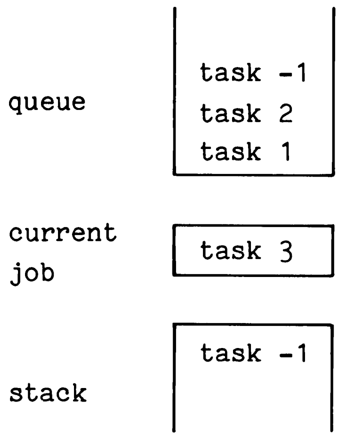
I. After approximately another Ø.2 seconds, task 4 is again put in the queue by the clock driver.



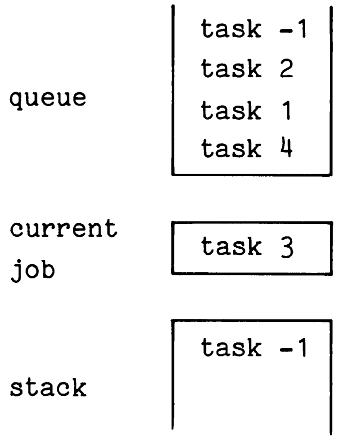
J. High priority task 4 interrupts the current task; task 2 is pushed onto the stack.



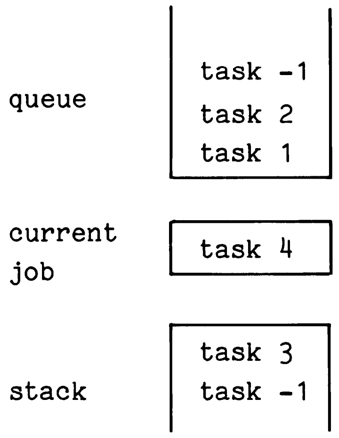
K. Task 4 puts itself in the clock queue and removes task 2 from the stack and places it back in the queue.



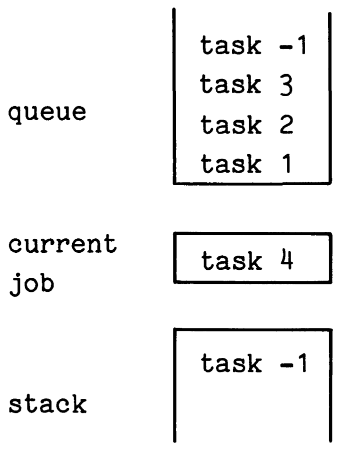
L. When task 4 exits, task 3 is popped from the queue and starts executing its loop.



M. After approximately Ø.2 seconds, high priority task 4 is entered at the front of the queue again.



N. Task 4 takes over once more; task 3 is pushed onto the stack.



O. Task 4 puts itself in the clock queue again and takes task 3 from the stack and puts it back on the queue. When task 4 completes, the Scheduler looks like diagram D.

RESET (Nonresident)

Examples:

```
   RESET #3
5Ø RESET #J+2
```

Syntax Form:

[line no.] **RESET #**expression

Descriptive Form:

[line no.] **RESET #**plun

Purpose:

To reset a sequential-access file, which is already OPEN for READ, to the beginning of that file.

Discussion:

The RESET command performs a CLOSE and OPEN for READ on the specified file. The file may then be read again. The file must be currently OPEN for READ.

If the peripheral logical unit number (PLUN) specified is associated with a paper tape reader (PR), RESET has no effect. The tape is not repositioned to the beginning of the tape, but is left in its current position.

Using the Command Syntax:

The expression for the PLUN is evaluated and rounded to an integer. If it evaluates to zero (the keyboard), a warning error is issued and nothing is done.

RETURN

Example:

```
650 RETURN
```

Syntax Form:

[line no.] **RETURN**

Purpose:

To return program control to the statement following a GOSUB command or to the statement that was about to be executed when an instrument event occurred.

Discussion:

The RETURN command has a variety of functions. In its simplest use, **the RETURN command signals the end of a subroutine.** When the command is executed, program control goes back to the command immediately following the GOSUB command that called the subroutine. The simple program below demonstrates this function. The subroutine starting at line 1000 sums the two values A and B, and places the result in variable X. The RETURN statement sends control back to line 120 (the statement following the GOSUB), and the result is printed at the terminal.

```
100 INPUT A,B
110 GOSUB 1000
120 PRINT X
 .
 .
 .
1000 X = A+B
1010 RETURN
```

RETURN also serves as the end of an interrupt routine. When an event occurs, control goes to the line number specified in the associated WHEN or SCHEDULE command. This transfer occurs (depending on the system priority) after completion of the command that is currently executing (or immediately if BASIC is in the idle mode). When the interrupt program is complete, the RETURN command sends program control back to the line that would have been executed next had the interrupt not occurred. If BASIC was in idle mode when the interrupt occurred, the RETURN puts BASIC back into the idle mode.

RETURN should be used in place of END or STOP when more than one task is executing or interrupts are allowed. Both END and STOP clear the Scheduler stack and queue, clear the clock queue, and cancel the action of any WHEN statements. RETURN leaves these structures intact to execute any remaining tasks and process pending interrupts.

[This command functions by returning control to the Scheduler with an indication that the current job is completed. The Scheduler then selects the new current job as described in Section 1.]

If the Scheduler stack and queue are empty when RETURN executes,

```
    READY
    *
```

is printed on the terminal to indicate that the system is in idle mode.

REWIND (Nonresident)

Examples:

```
    REWIND MT:
170 REWIND CT1:
```

Syntax Form:

[line no.] **REWIND** device name[constant]:

Descriptive Form:

[line no.] **REWIND** device name[drive number]:

Purpose:

To rewind a serial-tape device.

Discussion:

If the specified device is not capable of being rewound (such as a disk), the command is ignored.

Using the Syntax Options:

Since a tape device cannot be the system device, there is no default device. The device driver must be LOADed into memory before execution of the REWIND command. If the drive number is omitted, zero is assumed.

RUN

Examples:

```
RUN
RUN AS TASK N
```

Syntax Form:

RUN [**AS TASK** expression]

Descriptive Form:

RUN [**AS TASK** task number]

Purpose:

To start the execution of the program in memory at the lowest numbered line.

Discussion:

The RUN command transfers control to the program line in memory with the smallest line number. It automatically sets the program priority level to the default value of 5Ø. It also assigns a task number, either the optionally specified number or the default value of zero.

[Several other commands can be used to start a program executing. For instance, an OLD or a CHAIN command that designates a starting line number not only loads a program but also begins execution of it. When the OLD or CHAIN is a part of a program, the new program is assigned the same task number as the old program of which the OLD or CHAIN is a part. If the OLD or CHAIN is entered in immediate mode (has the immediate mode task number of 127), the task number of the program is reset to zero, the default task number of the RUN command.]

[It is desirable that programs not run as task 127 because task 127 cannot be ABORTed under program control. Only a Control-P, an immediate mode ABORT with no task number specified, or a fatal error in task 127 will halt it. That is why the system will not assign task 127 to programs started under the RUN, OLD, or CHAIN commands.]

[An immediate mode GOTO can also be used to start a program, but it does not give this protection. The program runs as task 127. In a debugging situation, this may be acceptable, since normal termination may not be expected and a Control-P is often used to halt a program being debugged. However, because of the task number assignment mechanism, **an immediate mode GOTO statement specifying the lowest numbered program line in memory is not the same as the RUN command.**]

Using the Syntax Option:

RUN must be executed in the immediate mode. It cannot be preceded by a line number.

The expression following the optional keywords **AS TASK** is rounded to an integer and used as the task number. It must evaluate to an integer between Ø and 126, inclusive. When the keywords and expression are omitted, a task number of zero is assumed.

SAVE (Nonresident)

Examples:

```
    SAVE "TEST.BAS"
    SAVE DK2:"SIGNAL"
130 SAVE CT:NM$,50,500
150 SAVE "ONE.LIN",100
    SAVE DL2:"MAILST" INTO 9
```

Syntax Form:

[line no.] **SAVE** [device name[constant]:][string expression] [**INTO** expression]
[,expression[,expression]]

Descriptive Form:

[line no.] **SAVE** [device name[drive number]:] [program file name]
[**INTO** number of blocks] [,line number [starting, line number ending]]

Purpose:

To save lines of program text on a peripheral device.

Discussion:

SAVE allows you to save all or part of the program that is in memory on a peripheral storage device.

Specifying the one or two optional line numbers allows you to SAVE only part of the program text that is in memory. If there is no text in the range of the line numbers specified (or no text in memory at all), a file is not created, and nothing is output to the device.

Since the optional line numbers in the SAVE command are expressions, they are **not** altered by the RENUM command.

If a line of text to be SAVEd is longer than 8Ø characters, a warning error is issued. The line in question should be corrected and the file REPLACEd. The OLD statement, used to load programs that have been SAVEd, does not accept lines longer than 8Ø characters.

For faster execution of segmented programs, use the OVLSAV command instead of SAVE and then use the OVLOAD command instead of OVERLAY.

Using the Syntax Options:

If no device is given, the system device is assumed. If the named device does not use the system device driver, its driver must be LOADed into memory before SAVE executes. (The keyboard, KB, may not be specified.) When the device is a serial tape device, no files may be OPEN on it. If the drive number is not specified, zero is assumed.

A file name is required for a file-structured device. (Non-file-structured devices are permitted to allow you to output a program to a paper-tape punch, PP). If no extension is present in the file name, .BAS is assumed.

If the target device is directory-structured (e.g., DK, DL, or DX) the **INTO** option can be used. The (rounded) expression following the keyword INTO stipulates the maximum number of blocks required by the file. When the INTO option is used, the first sufficient empty space on the target device is selected for the file. When the INTO option is not used; one half of the largest empty space on the target device is set aside for the file. In either case if the specified or default space exceeds the actual number of blocks needed for the file, the unused blocks are returned to an empty status. (The INTO option is not supported by SAVE VØ2-Ø1.)

When storing a program file on a nearly full disk, use the INTO option. Half the remaining free space may not be large enough for the file. In order to use all the available disk space, you will need to specify the required number of blocks rather than use the default.

The optional expressions are rounded to integers and used as line numbers. They must evaluate to numbers between 1 and 32767, inclusive. If one line number appears, only that line is SAVEd. When two line numbers are used, all program lines between and including those lines are SAVEd. When the line numbers are omitted, all the program lines in memory are SAVEd.

SCHEDULE (Nonresident)

Examples:

```
55Ø SCHEDULE GOSUB 9ØØ
15Ø SCHEDULE AFTER 36ØØ GOSUB 1ØØØ
7ØØ SCHEDULE AT "9:1Ø:ØØ" GOSUB 1ØØØ
16Ø SCHEDULE AFTER .333 WITH 53 AS TASK 1Ø GOSUB 5ØØ
 7Ø SCHEDULE AT "18:3Ø:45" WITH 5 GOSUB 33Ø
```

Syntax Form:

```
                        ┌AFTER expression       ┐
[line no.] SCHEDU       │                       │[ WITH expression][ AS TASK expression]
                        └AT string expression   ┘

                         GOSUB line number
```

Descriptive Form:

```
                          ┌AFTER number of seconds┐
[line no.] SCHEDULE       │                       │[ WITH priority level][ AS TASK task number]
                          └AT time specification  ┘

                           GOSUB line number
```

Purpose:

To schedule a subroutine for execution after a specified amount of time has elapsed or at a specified time.

Discussion:

SCHEDULE gives BASIC the ability to perform subroutines at predetermined times, depending on the priority of the system. The time can optionally be specified as either a time interval to wait before scheduling a subroutine or a specific time at which to schedule a subroutine.

[When the SCHEDULE command executes, it places the information given about the subroutine in the clock queue. After the specified time has elapsed or at the specified time of day, the clock driver removes the information from the clock queue and enters the subroutine in the Scheduler queue. The task starts executing according to the rules for the Scheduler as explained in Section 1.]

This means that the transfer to the subroutine does not take place until two conditions are met: 1) the specified time elapses or the specified time of day (according to the system time) is reached, and 2) the priority of the system is less than the priority assigned to the scheduled subroutine. (If no priority is specified, a default priority of 51 is assigned. Note that this is one greater than the default program priority of 5Ø.) When these conditions are met, program control passes to the subroutine. The system assumes the priority level of the subroutine until a RETURN is encountered, terminating the subroutine (or until it is interrupted by a higher priority routine). Then control returns to the command that was about to be executed when the transfer occurred and the priority of the system reverts to the level of the system prior to the transfer.

The timing is not exact. If a command is executing when the subroutine is scheduled, the command finishes executing before the priority comparison is made. If the currently executing command is a complex input/output statement or a complex math operation, it may be several seconds before even a high priority subroutine can take control.

Up to 24 routines can be scheduled at a time. If equal priority routines are ready for execution at the same time, the first routine scheduled is executed first.

The clock driver "CLK.SPS" must be in memory before the SCHEDULE command executes. If the system has no line frequency clock, executing SCHEDULE causes an error.

Once a routine has been entered in the clock queue, it can be removed from the clock queue by the UNSCHEDULE command. This means that the action of a SCHEDULE statement can be canceled by UNSCHEDULE if the specified time has not elapsed. END, STOP, and Control-P not only clear the clock queue but also clear the Scheduler as well, so all pending tasks and interrupts are cancelled.

NOTE

The SCHEDULE command cannot be used with the PDP 11/03 or the CP4165 standard line time clock. This command assumes a DEC KW11-L (or equivalent) line frequency clock.

Using the Syntax Options:

You can specify the time interval to wait either as a relative number of seconds from the time SCHEDULE executes or as an absolute time of day. When neither time specification is given, the subroutine is scheduled immediately.

An expression following the keyword **AFTER** specifies the time interval (in seconds) the clock driver is to wait before scheduling the subroutine. For example:

36ØØ	is one hour
6Ø	is one minute
.333	is about 1/3 of a second

The best resolution using the AFTER form is 1/6Ø of a second.

A string expression following the keyword **AT** specifies the time of day at which the clock driver is to schedule the subroutine. The string expression must evaluate to a string of the form:

"HH:MM:SS"

where:

HH	is the hour. It can be one or two digits representing an integer between **Ø** and **23**, inclusive.
MM	is the minute. It can be one or two digits representing an integer between **Ø** and **59**, inclusive.
SS	is the second. It can be one or two digits representing an integer between **Ø** and **59**, inclusive.

The best resolution using the AT form is 1 second.

When this form is used, the time-of-day specification is compared to the system time, and the difference between the two times is used to determine when to schedule the subroutine. For this reason, the system time should be set by the SETTIME command before SCHEDULE is executed. If the time of day specified is earlier (less) than the system time, the subroutine is scheduled immediately.

The optional keyword **WITH** and its expression specify the execution priority to be assigned to the subroutine. The expression is evaluated and rounded to an integer. It must be between Ø and 126, inclusive, where 126 has highest priority. If this is omitted, a priority of 51 is used (one higher than the default system priority of 5Ø).

The expression following the optional keywords **AS TASK** specifies the task number of the subroutine. This expression, when evaluated and rounded to an integer, must be between Ø and 126, inclusive. When this task specification is omitted, the task number used is the number of the currently executing task. However, when the AS TASK and task number are omitted from an immediate mode SCHEDULE statement, the subroutine is scheduled as task number Ø.

The line number following the keyword **GOSUB** indicates the starting line number of the subroutine.

Application Example:

The following example program demonstrates how the SCHEDULE command can be used to sample a waveform from an instrument every 3Ø seconds and write that waveform to a peripheral storage device.

```
1Ø REM -- LOAD CLOCK DRIVER
2Ø LOAD "CLK.SPS"
3Ø REM -- SCHEDULE THE SUBROUTINE
4Ø REM -- ASSUME PERIPHERAL DRIVER AND INSTRUMENT DRIVER ARE ALREADY
5Ø REM -- IN MEMORY, AND THE INSTRUMENT IS ATTACHED AS ILUN #2
6Ø REM -- FILE ON PERIPHERAL IS OPENED AS PLUN #1
7Ø SCHEDULE AFTER 3Ø GOSUB 2ØØØ
8Ø REM -- SOME OTHER PROGRAM COULD GO HERE
 .
 .
 .
```

```
1Ø9Ø REM -- THIS ROUTINE IS ENTERED EVERY 3Ø SECONDS
2ØØØ GET #1 FROM #2
2Ø1Ø REM -- SCHEDULE THE ROUTINE AGAIN
2Ø2Ø SCHEDULE AFTER 3Ø GOSUB 2ØØØ
2Ø3Ø REM -- RETURN TO THE MAIN PROGRAM
2Ø4Ø RETURN
```

SETDATE (Nonresident)

Examples:

```
    SETDATE "16-NOV-98"
100 SETDATE A$
250 SETDATE A$&B$(10)
    SETDATE DT
```

Syntax Form:

$$[\text{line no.}]\ \textbf{SETDAT} \left\{ \begin{array}{l} \text{array expression} \\ \text{string expression} \end{array} \right\}$$

Descriptive Form:

[line no.] **SETDATE** date specification

Purpose:

To set the system date.

Discussion:

The SETDATE command allows the user to specify the date to the BASIC monitor. Once the date has been set, it can be returned by the DATE command. Also, any files SAVEd, REPLACEd, OPENed FOR WRITE, or DEFINEd on a directory-structured device will have that date. If the system date is not set, the date is null.

If the system software is reloaded, the date is cleared.

Using the Command Syntax:

The system date may be set by either an array expression or a string expression. (SETDATE VØ2-Ø1 does not allow the array expression argument.) The specified date must be a valid date.

When the date is specified by an array expression, it must result in exactly three elements. The value of each element is rounded to an integer and used in this order:

first element	is the month. It must be between **1** and **12**, inclusive.
second element	is the day of the month. It must be between **1** and **31**, inclusive, and it must be a valid day for the month specified.
third element	is the year. It must be between **76** and **99**, inclusive.

When the date is specified by a string expression, it must evaluate to a string of the form:

"DD-MMM-YY"

where:

DD	is the day of the month. It must be one or two digits representing an integer between **1** and **31**, inclusive, and it must be a valid day for the specified month.
MMM	is the first three characters in the name of the month.
YY	is the year. It must be two digits representing an integer between **76** and **99**, inclusive.

SETTIME (Nonresident)

Examples:

```
    SETTIME "9:33:30"
100 SETTIME A$
    SETTIME
    SETTIME TM
```

Syntax Form:

```
[line no.] SETTIM [array expression ]
                  [string expression]
```

Descriptive Form:

```
[line no.] SETTIME [time specification]
```

Purpose:

To set the system time.

Discussion:

The SETTIME command allows the user to specify the time of day to the BASIC monitor. The system must have a line frequency clock (e.g., a DEC KW11-L line frequency clock or an equivalent) before SETTIME can set the system time. If the system has a line frequency clock but the time of day has not been set, the system time reflects the elapsed time since the system software was loaded.

The system time can be returned by the TIME command. In addition, the form of the SCHEDULE command that has the AT time-of-day specification uses the system time to calculate the time interval to wait before initiating a subroutine's execution by entering the subroutine into the Scheduler queue. When using this form of SCHEDULE, the SETTIME command should be executed prior to the SCHEDULE command. (The function and parts of the Scheduler are explained in Section 1.)

SETTIME can also be used to turn off the line frequency clock by executing the command with no arguments. This allows the user to eliminate the time-keeping routine during time-critical data processing. After the clock is turned off, the system time remains unchanged until another SETTIME command resets the time and turns the clock back on.

The system time is not automatically changed from 23:59:59 to Ø:Ø:Ø. Instead, the time continues to increment. You should reset the time after midnight.

If the system has no line frequency clock, executing SETTIME causes an error.

Using the Syntax Options:

Specifying the optional array expression or string expression sets the system time and turns on the line frequency clock if it has been turned off. It must be a valid time. (SETTIME VØ2-Ø1 does not allow the array expression argument.)

When the time is specified by an array expression, it must result in exactly three elements. The value of each element is rounded to an integer and used in this order:

first element	is the hour. It must be between Ø and 23, inclusive.
second element	is the minutes. It must be between Ø and 59, inclusive.
third element	is the hour. It must be between Ø and 59, inclusive.

When the time is specified by a string expression, it should evaluate to a string of the general form:

"HH:MM:SS"

where:

HH is the hour. It can be one or two digits representing an integer between **Ø** and **23**, inclusive.

MM is the minute. It can be one or two digits representing an integer between **Ø** and **59**, inclusive.

SS is the second. It can be one or two digits representing an integer between **Ø** and **59**, inclusive.

Other acceptable forms are "HH", "HH:", "HH:MM", and "HH:MM:"

Omitting the string expression turns off the line frequency clock.

SQUISH (Nonresident)

Examples:

```
150 SQUISH DL:
160 SQUISH DK: TO DK1:
170 SQUISH DX1:,VER
```

Syntax Form:

$$[\text{line no.}]\ \textbf{SQUISH}\ \text{device name}[\text{constant}]\textbf{:}[\textbf{TO}\ \text{device name}[\text{constant}]\textbf{:}]\left[\textbf{,}\begin{Bmatrix}\textbf{VER}\\ \text{string expression}\end{Bmatrix}\right]$$

Descriptive Form:

[line no.] **SQUISH** source device name[drive number]**:**[**TO** target device name[drive number]**:**]
[**,**bad block **VER**ification switch]

Purpose:

To compress the directory and files on the target device. All free (unused) blocks are then located in one area, following the files on the device.

Discussion:

As files are removed from a directory-structured device, empty spaces are created. These spaces appear as unused blocks when a directory of the device is displayed. These canceled files cause two problems: space is still required in the directory to note the location of the unused blocks and available free storage space is fragmented. The SQUISH command can delete these "unused" directory entries, compressing the remaining files into contiguous locations. This results in one larger area of free storage space, rather than several smaller areas. Any file with an extension of .BAD is not moved. (The .BAD extension should be reserved for signaling the location of damaged blocks.)

CAUTION

If power to the controller is interrupted while SQUISH is compressing the files on a device, all data on that device may be lost.

The command can also be used to transfer copies of all the files from one device to another device, leaving the source device unchanged. However, the target device is zeroed and given a new directory with the same number of segments as the source device, so any data on the target device prior to the execution of SQUISH is effectively deleted. This makes it unnecessary to initialize the target device with the ZERO command beforehand.

CAUTION

When SQUISHing the files from one device to another, any data previously stored on the target is lost by the SQUISHing operation.

SQUISH with the VER option is used to verify the source device itself before any files are moved or transferred. If any bad (physically damaged) blocks are found, the block numbers (in octal) are printed on the terminal. The SQUISH is not done if any bad blocks are found.

Using the Syntax Options:

The source device must be specified but the keyword **TO** and the target device are optional. If the target device is supplied, the source device's files are transferred to the target device. If the target device is omitted, the source device is also the target device so its files are moved to contiguous locations (compressed) on the device. If the target device is DY, the disk must be in double-density format. (See the FORMAT command.) This means that an RX02 (or equivalent) disk formatted for single density can be SQUISHed to a double-density format RX02 disk, but should not be SQUISHed to itself, using the DY driver. (If a single density disk is SQUISHed to itself with the DY driver, the data will remain in single density format; however, the number of free blocks will be set to the double density value. This disk will still be usable, but errors may result from the incorrect number of free blocks.)

The device(s) must be directory-structured. If either device does not use the system device driver its driver must be LOADed before SQUISH executes. When a driver number is omitted, zero is assumed.

If the optional keyword **VER** (or a string expression evaluating to "VER") is used, the source device is checked for bad blocks.

NOTE

The DL and DY drivers are not available in TEK SPS BASIC VØ2-Ø1.

Uses:

SQUISHing one disk to another disk is a convenient way of making a back-up copy of your software. (See Appendix B for details.)

STATUS (Nonresident)

Examples:

```
    STATUS
    STATUS LP:
250 STATUS DK1:"STAT.FIL"
470 STATUS SCHED
```

Syntax Form:

```
                   ┌device name[constant]:[string expression][,SCHED]┐
[line no.] STATUS  │string expression[,SCHED]                        │
                   └SCHED                                            ┘
```

Descriptive Form:

```
                   ┌device name[drive number]:[target file name][,SCHEDuler information flag]
[line no.] STATUS  │target file name[,SCHEDuler information flag]
                   └SCHEDuler information flag
```

Purpose:

To print the status of the system on the specified device or file.

Discussion:

STATUS outputs either general information about the system or the contents of the Scheduler. In either case, the output can be displayed on the terminal, printed on a device such as a line printer, or sent to an ASCII format file.

If a target file is named, that file must not already exist on the device. (The STATUS command opens, fills, and then closes the file.) The information in the file can later be output to a device such as a line printer or keyboard terminal with the COPY command.

When the keyword SCHED is omitted, the information that is provided includes:

1. The number of words of free memory. **For extended memory (XM) systems, the number of words of free array-storage memory is also displayed.**

2. The names of any peripheral and instrument drivers in memory plus the maximum number allowed.

3. The names of nonresident commands in memory plus the maximum number allowed. (Those auto-loaded are designated by the word AUTO.)

4. The maximum number of files that can be open at one time and the names of each OPEN file including the operation for which it is open and its peripheral logical unit number (PLUN).

5. The maximum number of instruments that can be attached at one time and the list of attached instruments. Given with each instrument is its instrument logical unit number (ILUN) and its interface number for IEEE 488 devices.

Below is sample output of the STATUS command when the keyword SCHED is omitted.

```
*STATUS
FREE MEMORY 3090 WORDS

DRIVERS IN MEMORY (6 PERIPH., 2 INSTR. MAX)
DK
KBG
DPO
IV

NONRESIDENT COMMANDS IN MEMORY (12 MAX)
SCHEDU (AUTO)
PRINT (AUTO)
STATUS (AUTO)
DIR
CHANGE (AUTO)

OPEN FILES (4 MAX.)
DKØ: FOO        IS OPEN FOR WRITE AS PLUN #1
DKØ: PWRFAL.DOC IS OPEN FOR READ AS PLUN #4
```

```
ATTACHED INSTRUMENTS (2 MAX.)
IVØ: IS ATTACHED AS ILUN #1
DPOØ: IS ATTACHED AS ILUN #2

READY
*
```

When the optional keyword SCHED is specified, the contents of the Scheduler queue, the current-job slot, and the Scheduler stack are output. (The operation and the structures of the Scheduler are described in Section 1.) Four pieces of information about each item on the Scheduler are printed:

1. Its task number.

2. Its priority.

3. The line it came from -- the line that put it on the stack or queue (or the currently executing line for the current job).

4. Its line number -- the line where execution begins or resumes when it becomes the current job. (The line number of the current job will always be the line number of the STATUS SCHED statement.)

The contents of the Scheduler is of interest only when a program is running. Below is a sample output of a STATUS SCHED statement and the program in which it was executed:

```
*RUN
                TASK    PRI      FROM      LINE
                  -1     -1         Ø         Ø
                   1     11        1Ø     2ØØØØ
QUEUE...           2     22        2Ø     2ØØØØ
CURRENT...         Ø     5Ø     1ØØØØ     1ØØØØ
STACK...           Ø     5Ø      1ØØØ      1Ø1Ø
                   Ø     5Ø       1ØØ       1ØØ
                  -1     -1         Ø         Ø

HI
 1
 2
READY
```

```
*LIST
1Ø SCHEDULE WITH 11 AS TASK 1 GOSUB 2ØØØØ
2Ø SCHEDULE WITH 22 AS TASK 2 GOSUB 2ØØØØ
1ØØ GOSUB 1ØØØ\PRINT "HI"
11Ø RETURN
1ØØØ GOSUB 1ØØØØ
1Ø1Ø RETURN
1ØØØØ STATUS SCHED
1ØØ1Ø RETURN
2ØØØØ PRINT TSK(Ø)
2ØØ1Ø RETURN
```

At the time the STATUS SCHED command executes, both line 1Ø and line 2Ø have put line 2ØØØØ into the queue. Line 1Ø SCHEDULEd the subroutine at line 2ØØØØ as task 1; line 2Ø SCHEDULEd it as task 2. Also, the GOSUBs in lines 1ØØ and 1ØØØ have caused lines 1ØØ and 1Ø1Ø, respectively, to be pushed onto the stack.

The two idle packets, which always remain in the Scheduler, have task numbers and priority numbers of -1. They, of course, came from no line number and have no line numbers, so zeros are assigned to their "come from" and line number values.

Notice that the next-to-the-last item on the stack, line 1ØØ, came from line 1ØØ. This is because the GOSUB command in line 1ØØ is followed by a command in the same line. When the subroutine called by the GOSUB in line 1ØØ exits, control returns to the PRINT command in line 1ØØ.

Using the Syntax Options:

When you omit both the optional driver name and target file name, the information is sent to the keyboard terminal. When the target device name is omitted but a target file is named, the system device is assumed. If the target device name is stipulated but the target file name is omitted, the device must not require a file name (e.g., a line printer). If the named device does not use the system device driver, its driver must be LOADed into memory before STATUS executes. If the drive number is omitted, zero is assumed.

Specifying or omitting the keyword **SCHED** determines what status information is output. When SCHED is used, the contents of the Scheduler is output. When SCHED is omitted, the general information about the system is output.

STOP

Example:

```
105 STOP
```

Syntax Form:

[line no.] **STOP**

Purpose:

To stop all program execution and return to idle mode.

Discussion:

The STOP command terminates a running program. It clears the Scheduler stack and queue of all tasks, returning the Monitor to idle mode. (The function and parts of the Scheduler are explained in Section 1.) STOP cancels the action of all WHEN statements and clears the clock queue. It also disables any INPREQ and ONERR conditions. However, all OPEN files remain OPEN, and all ATTACHed instruments are left ATTACHed.

The STOP command may appear anywhere in a program. Any number of STOP commands may be used.

If the command is executed in immediate mode, the message

STOP

is printed on the terminal. If the command is executed in program mode,

STOP AT LINE XXX

is printed, where XXX is the line number of the STOP command.

STOP differs from END in the printing of this message and in that file are left OPEN. END CLOSEs any OPEN files.

Since it clears the Scheduler, executing STOP halts all tasks, not just the one in which it appears. To halt the current task and only that task, use the ABORT command without specifying a task number. This terminates the currently executing task.

SYSBLD (Nonresident)

Example:

```
SYSBLD
```

Syntax Form:

[line no.] **SYSBLD**

Purpose:

To allow the user to define the contents of the "SYSBLD.DEF" file. This file is used by Resident BASIC to set the system parameters at initialization.

Discussion:

After BASIC is loaded, no dialog takes place between the user and the Monitor to obtain the parameters to set the capabilities and size of the system. Instead, the initialization routine searches for a file named "SYSBLD.DEF" on the system device. If the file is not there, an internal list of default parameters is used to initialize the system. If the file is there, the user-defined parameters in the file are used.

The SYSBLD command lets you create or change the "SYSBLD.DEF" file. Then, the parameters in "SYSBLD.DEF" are used to initialize the system the **next** time BASIC is loaded from that disk and every time after that until the file is CANCELed or SYSBLD is executed again to change them.

To create the file, SYSBLD displays several questions on the terminal and records your responses. Depending on the question, acceptable answers are a Y or N, a number, or a file name -- followed by a carriage return. In all cases a carriage return by itself is also acceptable and means that you choose the default answer. **The default answer is given with each question.**

After the questions are answered, if a "SYSBLD.DEF" file already exists, it is canceled before the new file is written onto the system device. From then on, whenever you load BASIC from this disk, the parameters you defined using the SYSBLD command initialize the system.

If the parameters you set require a system to be too large, initialization will fail when you reboot. Before it dies, however, the Monitor will cancel the "SYSBLD.DEF" file and display the following message:

```
REQUESTED SYSTEM EXCEEDS AVAILABLE MEMORY.
ATTEMPTING TO CANCEL 'SYSBLD.DEF' FILE FROM SYSTEM DEVICE.

RELOAD SOFTWARE.
```

Since the faulty "SYSBLD.DEF" file is canceled, you can reload BASIC and execute SYSBLD again.

Below are the questions SYSBLD asks and a discussion of the possible answers. The questions, printed in bold, are just as they are displayed on the terminal.

RETAIN IEEE 488 (GPIB) CAPABILITIES (Y,N,CR, DEFAULT IS Y)?

The resident portion of the IEEE 488 code is a collection of routines that handle interrupts and routines that are commonly used nonresident commands and drivers to communicate with IEEE 488 devices. If you delete this by responding with an N, you will not be able to use a nonresident IEEE 488 driver. If you have no need of an IEEE 488 driver, deleting it saves between 650 and 880 words of controller memory depending on the version of the BASIC Monitor.

RETAIN STRING FUNCTIONS (Y,N,CR, DEFAULT IS Y)?

The string functions available in TEK SPS BASIC are explained in Section 5. If your answer to this question is an N, all string functions are deleted. This saves approximately 360 words of memory. Strings can still be used, but string functions cannot.

RETAIN GRAPHICS CAPABILITIES (Y,N,CR, DEFAULT IS Y)?

All graphics commands access common routines in Resident BASIC. If these resident routines are deleted, graphics modules cannot be used. The affected commands include all the commands in the Graphics Package and certain instrument-specific graphics commands such as TDPLOT and ADPLOT. If you have no need of any graphics routines, answering with an N saves about 11ØØ to 13ØØ words of controller memory depending on the version of the BASIC Monitor.

HOW MANY WORDS DO YOU WANT AS A PATCH AREA? (DEFAULT IS Ø)?

TEK SPS BASIC gives you the ability to alter Resident BASIC, nonresident commands, and drivers. These modifications are done by the PATCH files when software updates are released via the SPS Programming Update. If a nonzero value should be entered here, it will be supplied with the other information in the SPS Programming Update. Unless this situation applies, use the default answer -- a Ø or just a carriage return. Appendix D contains software patching information.

HOW MANY PERIPHERAL LOGICAL UNIT NUMBERS DO YOU WANT? (DEFAULT IS 6)?

Each file on a file-structured device and each peripheral device (such as a line printer) OPEN at any one time requires a unique peripheral logical unit number (PLUN). The number of peripherals you can OPEN at a time is limited by the number you supply here. If you want to use a line printer and have two disk files OPEN at the same time, you need at least three PLUNs. However, 17 words of memory are required for each PLUN requested. You can save memory by specifying only the number of PLUNs you are actually going to OPEN at once. See the OPEN command description for more information.

Even if you answer with a Ø to this question, PLUNs Ø and 1 will still be set up. This is because PLUN Ø is required for the keyboard and some commands, like COPY, need an extra PLUN to execute.

HOW MANY INSTRUMENT LOGICAL UNIT NUMBERS DO YOU WANT? (DEFAULT IS 8)?

Your answer here determines the number of instruments you can ATTACH at one time. Eleven words of memory are required for each instrument logical unit number (ILUN) requested. You can save controller memory by requesting only the number of ILUNs you need. If you have only two instruments, 2 is a good answer here.

HOW MANY PERIPHERAL DRIVERS DO YOU WANT TO USE AT ONCE? (DEFAULT IS 4)?

Only four words of memory are required at initialization for each peripheral driver you request, but remember that additional room will be needed for the driver when it is LOADed. The number you give here (or the default of 4) does not include the keyboard terminal driver and the system device driver that are loaded at initialization. If your only other peripheral is a line printer, 1 is a sufficient answer.

HOW MANY INSTRUMENT DRIVERS DO YOU WANT TO USE AT ONCE? (DEFAULT IS 4)?

Each different type of instrument requires its own driver. Four words of memory are needed for each instrument driver you request (plus room for the driver when it is LOADed). If you need three different instrument drivers in memory at the same time, answer this question with a 3.

HOW MANY NONRESIDENT COMMANDS DO YOU WANT RESIDENT AT ONCE? (DEFAULT IS 6)?

Your answer to this question limits the number of nonresident commands you can have in memory at one time. Six words of memory will be needed for each nonresident command requested (plus room for the command itself when it is brought in). A carriage return response allows six nonresident commands to be resident at once. If your applications require more nonresident commands per program and if each command executes often, you should consider giving a number large enough to meet your programs' needs. If you have the memory space to spare, this will prevent the continual loading and releasing of commands and, therefore, speed execution.

ENTER LINE CLOCK FREQUENCY IN HZ. (DEFAULT IS 6Ø)?

If you are using TEK SPS BASIC in a location where the electrical current is 50 Hz, enter 50. Otherwise use the default value (a carriage return or 60). A meaningless answer such as 75 will be accepted, but your system will not keep time correctly.

```
WHICH KEYBOARD DRIVER DO YOU WANT RESIDENT?
      (TYPE                         FILENAME
      GRAPHICS . . . . . . . .      KBG (DEFAULT)
      NO GRAPHICS. . . . . . .      KBN
      TV . . . . . . . . . . .      KBT
      EHANCED GRAPHICS . . . .      KBE)  ENTER FILE NAME:?
```

If you answered with a Y (or carriage return) to the graphics capabilities question, answer this with KBG or KBE. Though slower than KBG, KBE allows the high resolution graphics available with a TEKTRONIX 4014 Computer Display Terminal. KBE is compatible with any TEKTRONIX 4010-Series terminal, but it provides the high resolution graphics only when used with a 4014 terminal equipped with the Enhanced Graphics Module. If your answer to the graphics question was an N, save memory by answering KBN or KBT. Use the TV mode keyboard driver if you have a TV type terminal. (The enhanced graphics option is not supported by TEK SPS BASIC VØ2-Ø1.)

For extended memory (XM) systems, one additional question is asked:

```
HOW MANY 1K-WORD SEGMENTS DO YOU WANT TO RESERVE
FOR ARRAY STORAGE IN EXTENDED MEMORY?
(DEFAULT IS 96 OR MAXIMUM EXTENDED MEMORY IN SYSTEM)?
```

Up to 96K words of extended memory can be added to the standard 28K words of memory. In XM systems this extended memory can be used for array storage, as a peripheral accessed by the VM driver, or both. The answer to this question determines how many 1K-word segments of extended memory will be allocated for array storage. An answer of a carriage return sets aside all of your extended memory for array storage. To reserve some extended memory for use as a peripheral, specify a number smaller than the total number of 1K-word segments of your extended memory. If you don't use the VM driver, answer this with a carriage return.

The following program will display the contents of your present "SYSBLD.DEF" file for a standard memory system. Each answer to all but the keyboard driver question is stored as one word -- the first three as ASCII characters, the next seven as integers. The answer to the keyboard driver question is stored as nine ASCII bytes, but this program reads only the first three. Because there are no data descriptors or delimiters between data items, READU is used to read the data from the file.

```
10 REM READ PARAMETERS FOR 'SYSBLD.DEF'
20 REM NO DELIMITERS OR DATA DESCRIPTORS IN FILE
30 REM USE READU COMMAND
40 OPEN #1 AS "SYSBLD.DEF" FOR READ
50 DIM A$(3)
60 INTEGER A(6)
70 REM FIRST 3 STORED AS 2-BYTE STRINGS
80 FOR N=0 TO 2
90 READU #1,A$(N)=2
100 NEXT N
110 REM NEXT 7 STORED AS INTEGERS, READ IN AS ARRAY
120 READU #1,A
130 REM LAST PARAMETER IS STRING
140 READU #1,A$(3)=3
150 CLOSE #1
160 REM LABEL AND PRINT CONTENTS
170 DIM S$(3),T$(6)
180 S$(0)="GPIB
190 S$(1)="STRING FUNCTIONS
200 S$(2)="GRAPHICS
210 S$(3)="KEYBOARD DRIVER
220 T$(0)="PATCH AREA
230 T$(1)="PLUNS
240 T$(2)="ILUNS
250 T$(3)="PERIPHERAL DRIVERS
260 T$(4)="INSTUMENT DRIVERS
270 T$(5)="NONRESIDENT COMMANDS
280 T$(6)="CLOCK FREQUENCY
290 PRINT,"CONTENTS OF 'SYSBLD.DEF'
300 PRINT
310 FOR N=0 TO 2
320 PRINT,S$(N);TAB(36);A$(N)
330 NEXT N
340 FOR N=0 TO 6
350 PRINT,T$(N);TAB(35);A(N)
360 NEXT N
370 PRINT,S$(3);TAB(36);A$(3)
380 RETURN
```

From a loop, line 90 reads in two ASCII bytes for the answer to each of the first three questions and stores them in a string array, A$. Then line 120 reads an integer array, A, to obtain the answers to the next seven questions. Finally, line 14Ø reads in the keyboard driver file name as the last item in the string array. The rest of the program produces a labeled display of the file's contents. If you prefer, you could simply PRINT the two arrays that hold the information, A$ and A, by replacing lines 16Ø through 37Ø with:

```
          16Ø  PRINT A$,A
```

For extended memory (XM) systems, the program should be modified to read and report the answer to the array storage question. This answer is stored as an integer following the keyboard driver name, with a null byte in between. One way to alter the program is to add these four lines:

```
          144 READU #1,B$=7
          146 INTEGER B(Ø)
          148 READU #1,B
          375 PRINT, "XM ARRAY STORAGE"; TAB(35);B(Ø)
```

The first three lines read the rest of the keyboard driver file name and the null byte, create another integer variable, and read the XM parameter. Line 375 prints the answer.

TIME (Nonresident)

Examples:

```
250 TIME T$
310 TIME A(0:2)
100 TIME T
    TIME
```

Syntax Form:

```
                    ┌simple numeric variable┐
[line no.] TIME     │array                  │
                    └string variable        ┘
```

Descriptive Form:

```
                    ┌target variable        ┐
[line no.] TIME     │target array           │
                    └target string variable ┘
```

Purpose:

To return the system time.

Discussion:

The TIME command either returns the current system time in the specified argument or prints the time on the terminal if the argument is omitted. When the argument is supplied, the time is returned either as three array elements or as a string, depending on the type of the specified variable.

The system time can be set by the SETTIME command. If the time is not set, the system time reflects the time interval since the system software was loaded. The system time does <u>not</u> wrap around at midnight from 23:59:59 to Ø:Ø:Ø, but it continues to increment. If the time exceeds 24 hours, it should be reset by the SETTIME command.

If the system does not have a DEC KW1-L or equivalent line frequency clock, executing TIME (or SETTIME) causes an error.

When the time is returned as three array elements, they are stored in the array in this order:

first element	hour (Ø-23 or higher if the system time is not reset after 23:59:59)
second element	minute (Ø-59)
third element	second (Ø-59)

When the time is returned in a string variable, it is of the form:

HH:MM:SS

where:

HH	is the hour (Ø-23 or higher if the system time is not reset after 23:59:59)
MM	is the minute (Ø-59)
SS	is the second (Ø-59)

Using the Syntax Options:

Specifying either a simple (not subscripted) numeric variable or an array returns the time in an array. If a simple numeric variable is used, it is auto-dimensioned to a three-element integer array. If an array is used, it must be dimensioned or zoned to exactly three elements.

Specifying a string variable returns the time in that string.

Omitting the argument prints the time on the terminal in the string variable format.

Application Example:

The TIME command can be used to print the current time on program runs. A simple method is to return the time as a string and PRINT it. For example:

```
100 TIME T$
110 PRINT #N,"TIME:";T$
```

where N is assumed to be the peripheral logical unit number (PLUN) of the line printer.

TIME can also be used to calculate the running time of a program or subroutine to the nearest second. For example:

```
100 TIME T1
110 REM FIRST LINE OF ROUTINE
 .
 .
 .
800 REM LAST LINE OF ROUTINE
810 TIME T2
820 REM CALCULATE RUNNING TIME
830 REM ASSUME LESS THAN AN HOUR
840 REM SUBTRACT SECONDS, BORROW IF NECESSARY
850 IF T2(2)>=T1(2) THEN 890
860 IF T2(1)=0 THEN T2(1)=60
870 T2(1)=T2(1)-1
880 T2(2)=T2(2)+60
890 T2(2)=T2(2)-T1(2)
900 REM SUBTRACT MINUTES, BORROW IF NECESSARY
910 IF T2(1)>=T1(1) THEN 930
920 T2(1)=T2(1)+60
930 T2(1)=T2(1)-TI(1)
940 PRINT "RUNNING TIME",T2(1);" MINUTES",T2(2);" SECONDS"
950 RETURN
```

The first statement of the routine being timed returns the beginning time in the auto-dimensioned array, T1. Then, just before the routine terminates, the ending time is returned in another auto-dimensioned array, T2. The running time is the difference of the two times, but you cannot just subtract one from the other. Here we assume the running times will be less than an hour and simply subtract, first the seconds and then the minutes, borrowing when necessary. However, you could, instead, convert both times to the total number of seconds, subtract, and convert the answer. The result is printed on the terminal in the form:

```
RUNNING TIME: MM MINUTES   SS SECONDS
```

UNSCHEDULE (Nonresident)

Examples:

```
350 UNSCHEDULE GOSUB 3500
490 UNSCHEDULE TASK 4
510 UNSCHEDULE ALL
```

Syntax Form:

```
                     ( GOSUB line number )
[line no.] UNSCHE    { TASK expression    }
                     ( ALL                )
```

Descriptive Form:

```
                         ( GOSUB line number            )
[line no.] UNSCHEDULE    { TASK task number             }
                         ( ALL scheduled line numbers   )
```

Purpose:

To remove a subroutine from the clock queue, preventing it from being scheduled for execution (entered into the Scheduler queue).

Discussion:

The SCHEDULE command places into the clock queue the information needed to schedule a subroutine. Then, after the specified time has elapsed or the specified time of day is reached, the clock driver enters this subroutine into the Scheduler queue, scheduling it for execution. The UNSCHEDULE commands allows you to remove the subroutine from the clock queue before the specified time has passed, preventing the subroutine from being scheduled for execution. This command has no effect on subroutines already in the Scheduler queue. (The function and parts of the Scheduler are explained in Section 1.)

The clock driver "CLK.SPS" must be in memory when UNSCHEDULE executes.

Using the Syntax Options:

The line number following the keyword **GOSUB** should be to the beginning line number of a subroutine specified in a previously executed SCHEDULE command. When this specification is used, all entries associated with that line number are removed from the clock queue. If no entries have that line number, no error results.

The expression following the keyword **TASK** represents a task number. When evaluated and rounded to an integer, it must be between Ø and 126, inclusive. When this specification is used, all entries associated with that task number are removed from the clock queue.

The keyword **ALL** clears the clock queue.

Application Example:

For demonstration, the following program schedules a routine for execution 3Ø seconds from the time line 5Ø is executed. If the processing in lines 6Ø to 17Ø is completed before the 3Ø seconds have elapsed, the UNSCHEDULE command is used to prevent the routine from executing.

```
1Ø REM -- LOAD THE CLOCK DRIVER
2Ø LOAD "CLK.SPS"
3Ø REM -- SCHEDULE THE SUBROUTINE AT LINE 1ØØØ FOR
4Ø REM -- EXECUTION 3Ø SECONDS FROM NOW
5Ø SCHEDULE AFTER 3Ø GOSUB 1ØØØ
6Ø REM -- PROGRAM GOES HERE
 .
 .
 .
17Ø REM -- PROGRAM COMPLETE IN LESS THAN 3Ø SECONDS
18Ø UNSCHEDULE 1ØØØ
19Ø RETURN
98Ø REM -- EXECUTE THIS ROUTINE IF PROGRAM TAKES
99Ø REM -- LONGER THAN 3Ø SECONDS
1ØØØ PRINT "TIME HAS ELAPSED"
 .
 .
 .
11ØØ REM -- RETURN TO THE INTERRUPTED PROGRAM
111Ø RETURN
```

VARTST (Nonresident)

Examples:

```
160 VARTST B(I),"20",Q
190 VARTST A$,J+2,X
750 VARTST B$&C$,"1000",Y
```

Syntax Form:

$$\text{[line no.] } \textbf{VARTST} \left\{\begin{array}{l}\text{expression}\\ \text{string expression}\end{array}\right\}, \left\{\begin{array}{l}\text{expression}\\ \text{string expression}\end{array}\right\} \text{,variable}$$

Descriptive Form:

$$\text{[line no.] } \textbf{VARTST} \left\{\begin{array}{l}\text{decimal value to be tested}\\ \text{octal value to be tested}\end{array}\right\},$$

$$\left\{\begin{array}{l}\text{decimal specification of bits to be tested}\\ \text{octal specification of bits to be tested}\end{array}\right\} \text{,target for test result}$$

Purpose:

To test if any of the bits set in the second value are also set in the first value.

Discussion:

The VARTST command converts the results of the first two arguments into 16-bit binary integers and then compares these two integers. If any of the bits set in the first converted value are also set in the second, VARTST returns a 1 in the third argument. If none of the same bits are set, VARTST returns a Ø in the third argument. (A bit is set if it is a 1 and not set if it is a Ø.)

VARTST does not alter the values of any variables that may appear in the numeric expressions or string expressions used as the first two arguments. It changes only the value of the third argument -- to a 1 or Ø.

Using the Syntax Options:

The first argument is the value tested; the second argument is the value to which the first is compared.

If either of these arguments is a string expression, it is interpreted as an octal value. This string expression must evaluate to a string of no more than eight octal digits. However, only the lower 16 binary digits (bits) are used in the comparison. If either of the first two arguments is an expression, it is truncated to a 32-bit integer and again only the lower 16 bits are used.

Application Example:

This command can be used to test the contents of an integer array N for odd elements. In the sample routine below, any odd values are then made even by incrementing them by 1.

```
1000 FOR I=0 TO SIZ(N)-1
1010 VARTST N(I),"1",R
1020 N(I)=N(I)+R
1030 NEXT I
```

When an array element N(I) is even, R is 0. When an element is odd, a 1 appears in the last bit, so R equals 1. In line 1020 adding a 0 to an even number, keeps it even; adding a 1 to an odd number makes it even.

Also, the result of a VARTST statement can be used to direct program flow. For example:

```
100 VARTST A$,B$,R
110 GOTO R+1 OF 1000,2000
      .
      .
      .
```

Here, a program branches one of two ways depending on if any bit set in the octal value in A$ is also set in the octal value in B$.

VERSION (Nonresident)

Examples:

```
65 VERSION DK1:"UNLOG"
75 VERSION "MOVE.SPS",N$
   VERSION "WAIT"
   VERSION BASIC
```

Syntax Form:

```
                   ([device name[constant]:][/{F}[,]] string expression)
                   (                          {R}                      )
[line no.] VERSIO <         [,string variable]                          >
                   (BASIC [,string variable]                            )
```

Descriptive Form:

```
                    ([device name[drive number]:] [/forward or reverse switch[,]])
[line no.] VERSION <          driver or command name [,target string variable]    >
                    (BASIC monitor [,target string variable]                      )
```

Purpose:

To obtain the version and release numbers of a driver, a nonresident command or, the BASIC monitor.

Discussion:

All TEK SPS BASIC modules contain version and release information so that updates can be recognized even though the command or driver name is the same. This command is used to obtain the version and release numbers of drivers, nonresident commands, and the BASIC monitor. (The version and release of Resident BASIC is also displayed each time the system is booted).

The information is returned as a string with the format:

```
Vxx-yy or Vxx-yyXM
```

where xx is the version number, yy is the release number, and XM indicates that it is the extended memory version. It is output to the terminal unless the optional string variable is included.

Releases of nonresident modules are independent of other modules, including Resident BASIC. However, all nonresident modules with a given version number are compatible with all Resident Monitors with the same version number but not with Monitors with a different version number.

Using the Syntax Options:

The named device is the peripheral on which the module is stored. If the device name is omitted, the system device is assumed. If the named device does not use the system device driver, its driver must be LOADed into memory before VERSION is executed. When the drive number is omitted, zero is assumed.

[The **/F** or **/R** switches (Forward or Reverse) may be specified for a serial tape device. The switch indicates the direction of the tape movement when searching for the file. If the switch is omitted, the tape is rewound before a forward search is made. When used with other peripherals, the switch is ignored.]

The string expression is used as a file name of a nonresident command or driver. If the file name contains an extension, it must be .SPS. When the name of a module is specified, the version and release numbers of that nonresident command or driver are returned. When the keyword **BASIC** is specified, the version and release numbers of the BASIC monitor are returned. (The keyword BASIC is not supported by VERSION VØ2-Ø1.)

Using the optional string variable returns the information in the specified string. Omitting it sends the information to the terminal.

WAIT (Nonresident)

Examples:

```
150 WAIT
160 WAIT 600
```

Syntax Form:

[line no.] **WAIT** [expression]

Descriptive Form:

[line no.] **WAIT** [number of milliseconds]

Purpose:

To halt program execution until a keyboard interrupt occurs, or a specified amount of time has elapsed.

Discussion:

The WAIT command produces either a timed or untimed pause in execution. When used with the optional expression, it causes the program to wait the stated number of milliseconds. For example:

```
190 WAIT 500
```

halts processing for one-half second.

Omitting the expression produces a pause of indefinite length and enables an interrupt from the keyboard. Execution halts until a character is typed at the terminal. Any printing or control character, except Control-P, may be typed. (Control-P will stop the program completely.)

If an instrument interrupt occurs while the WAIT statement (in either of its forms) is executing, the task associated with the event is scheduled (entered in the Scheduler queue), but no further processing occurs until the WAIT command has completed execution. (The function and parts of the Scheduler are explained in Section 1.)

This command is not designed to provide precise timing but to cause a wait of sufficient length for a particular event to occur. Its accuracy is dependent upon other operating system functions which are allowed to take place concurrently such as input/output or interrupt handling. If the number of these conflicting functions is minimized, the timing of the WAIT command can be fairly accurate. However, the time that the system takes to switch from one command to another cannot be readily determined. It is assumed that the time required to switch from the previous command to the WAIT command, plus the time taken to evaluate the expression, is one millisecond. Therefore, the value of the expression is decremented by one (1) before the timing loop is executed.

Using the Syntax Option:

The optional expression specifies the approximate number of milliseconds (1/1000 of a second) that program execution WAITs. The expression, when evaluated, is rounded to an integer. If the expression is omitted, the resulting untimed pause must be terminated by a keyboard interrupt.

Application Example:

Sometimes it is necessary to stop a program long enough to adjust an incoming signal, change equipment setup, or load a new disk. The WAIT command provides this untimed pause. For example:

```
200 PRINT "PRESS RETURN WHEN READY TO CONTINUE"
210 WAIT
```

would halt a program until a key on the keyboard is pressed.

WAVEFORM

Examples:

```
14Ø WAVEFORM AA IS A,IA,HA$,VA$
15Ø WAVEFORM W1 IS B1(511),IB,HB$,VB$
```

Syntax Form:

$$\text{[line no.] }\textbf{WAVEFORM}\left\{\begin{array}{l}\text{simple numeric variable}\\ \text{waveform}\end{array}\right\}\textbf{ IS}$$

$$\left\{\begin{array}{l}\text{simple numeric variable(expression[,expression])}\\ \text{array[(expression[,expression])]}\end{array}\right\},$$

numeric variable, simple string variable,simple string variable

Descriptive Form:

$$\text{[line no.] }\textbf{WAVEFORM}\left\{\begin{array}{l}\text{simple numeric variable}\\ \text{waveform}\end{array}\right\}\textbf{ IS}$$

$$\left\{\begin{array}{l}\text{simple numeric variable(first dimension[,second dimension])}\\ \text{array[(first dimension[,second dimension])]}\end{array}\right\},$$

data sampling interval,horizontal units,vertical units

Purpose:

To associate a name with an array, a related data sampling interval, and units string variables, for convenience of computation.

Discussion:

A waveform is a variable name associated with a data array, a data sampling interval variable, and two string variables for horizontal and vertical units information. The assumption here is that the array elements represent a digitized signal and that the data sampling interval (DSI) is the time between the array's data elements. The first string variable is for the measurement's horizontal units (typically "S" for seconds), and the second string variable is for its vertical units (typically "V" for volts). The array, DSI variable, and the units string variables may be referenced by other commands besides the WAVEFORM command associates them. They may be assigned values before and after the association is made. However, care should be used to insure that the DSI variable is not assigned a negative value by any prior or shared use of that variable.

The advantage of using a waveform rather than an array is that automatic units processing is provided with waveform operations. Once a waveform has been created, it can be used in almost any expression where an array is valid. For waveform processing rules and results, see the table "Arithmetic Operations With Waveforms" in Section 2. Also, see the LET command discussion for information on the results of waveform assignments.

After a waveform has been declared, the array and associated variables can still be referenced individually, without affecting the other variables. For example, the array can be used as the destination of an arithmetic statement and not cause any change to the three associated variables. The array can even be DELETEd and redimensioned to different specifications without deleting the waveform. Only when the waveform name is specified are the other variables altered.

Zones may **not** be used in conjunction with the waveform name. To use the zone feature, the associated array name must be used instead.

Waveforms are removed from memory by the DELETE statement. When you do this, the waveform is deleted, so the array and variables are dissociated from each other. The individual variables and the array are not deleted, however.

Using the Syntax Options:

The first argument is the waveform name. Only a simple numeric variable (not an array element) or a previously declared waveform variable may be specifed here. If a waveform variable is used, it must be redefined with the same variables and string variables named in its earlier declaration. If the array specifications are included, they must not change the array's previously declared dimensions unless the array has been DELETEd since the last time it was declared.

The arguments following the keyword **IS** name the four components of the waveform association: the array, the DSI, the horizontal units, and the vertical units.

The array, the first of these four arguments, can be specified by either a simple numeric variable (not an array element) or an array variable. If a simple numeric variable is used, it must be explicitly dimensioned to an array here by supplying its dimension specifications. These dimension specifications -- the one, or optionally two, expressions enclosed in parentheses -- are rounded to integers and used to define an array in the same manner as the DIM command expressions are used. The simple numeric variable is dimensioned to a floating-point array before the waveform association is established. If an array is given, its dimension specifications may be restated, but they may not be changed unless the array is DELETEd first. The array may be either floating-point or integer.

The second of the four parts of a waveform is the DSI variable. It is followed by the two units string variables which must be simple string variables (not string array elements). The first string variable is assumed to hold the horizontal units; the second, the vertical units.

These four variables need not be unique to a single WAVEFORM command. For instance, the same DSI variable and units string variables may be used in several different WAVEFORM statements. However, if the contents of a DSI variable or a units string variable changes for one of the waveforms, it changes for any other waveforms that use this same variable.

WHEN (Nonresident)

Examples:

```
500 WHEN #J HAS T$ AT N GOSUB 2000
600 WHEN #3 HAS "ACQ" GOSUB 5050
700 WHEN #2 HAS "CB1" AT 100 AS TASK 2 GOSUB 1000
800 WHEN @0 HAS "SRQ" GOSUB 3000
```

Syntax Form:

$$[\text{line no.}]\ \textbf{WHEN} \left\{\begin{matrix}\#\\ @\end{matrix}\right\} \text{expression } [\textbf{HAS} \text{ string expression}]\ [\textbf{AT} \text{ expression}]$$

$$[\textbf{AS TASK} \text{ expression}]\ \textbf{GOSUB} \text{ line number}$$

Descriptive Form:

$$[\text{line no.}]\ \textbf{WHEN} \left\{\begin{matrix}\#\text{ilun}\\ @\text{IEEE 488 interface number}\end{matrix}\right\} [\textbf{HAS} \text{ driver-dependent interrupt specfication}]$$

$$[\textbf{AT} \text{ priority level}]\ [\textbf{AS TASK} \text{ task number}]\ \textbf{GOSUB} \text{ line number}$$

Purpose:

To allow change in program flow based on an instrument interrupt.

Discussion:

This powerful command gives TEK SPS BASIC the ability to change the normal flow of program execution if an event (instrument interrupt) occurs. It allows BASIC to perceive and respond to the specified instrument interrupt. After the WHEN executes, if the specified event occurs, control transfers to the specified subroutine -- a user-written interrupt routine -- as soon as the system priority is lower than the specified priority. These priority comparisons are made only at the end of the execution of each command of the currently executing routine; routines can be interrupted, commands cannot.

The driver for the specified instrument or interface must be in memory when the WHEN command is executed. The instrument must be connected and powered up.

WHEN stores the interrupt information in the required driver. The information remains there until the proper IGNORE command removes it or the task associated with the WHEN is ABORTed. However, some or all of this information can be modified (overwritten) by executing another WHEN statement. Because the information is stored, BASIC can respond to more than one occurrence of the same event. (STOP, END, or Control-P nullifies the actions of **all** WHEN commands.)

The transfer to the subroutine does not take place until two conditions are met: 1) the specified event occurs, and 2) the priority of the system is less than the priority specified in the WHEN command. (If no priority is specified, an instrument default priority is assigned. This value may be found in the instrument driver manual.) When these conditions are met, program control passes to the subroutine. The system assumes the new priority level of the event until a RETURN statement is encountered. At that time, the priority of the system reverts to the level the system was operating at before the transfer took place. Control returns to the command that was about to be executed when the transfer occurred.

[In terms of the action in the Scheduler, when a WHEN command executes, the interrupt information is stored in the required driver, permitting BASIC to recognize the given interrupt. When the event occurs, a packet with the stored line number, priority number, and task number is entered into the Scheduler queue. As soon as this packet's priority is higher than the current job's priority, the current job is interrupted and pushed onto the Scheduler stack. Then the interrupt routine's packet is popped off the Scheduler queue and the interrupt routine starts executing. When it finishes (a RETURN is encountered), the interrupted routine is popped off the stack and resumes executing. (The function and parts of the Scheduler are explained in Section 1.)]

The manual for the instrument driver being used gives complete documentation on possible interrupts and the valid interrupt specification strings.

Using the Syntax Options:

Any expression used in a WHEN command is **rounded** to an integer.

The expression following the pound sign (#) or the at sign (@) indicates the instrument or interface from which an interrupt will be recognized. If a pound sign (#) is used, the expression is the instrument logical unit number (ILUN) to which an instrument is ATTACHed. When evaluated, it must be between 1 and n, inclusive, where n is the number of ILUNs specified at system initialization (default value of eight). If an at sign (@) is used, the expression represents the number of the IEEE 488 interface through which more than one instrument may be controlled. In this case, instead of an instrument-specific driver, the low-level IEEE 488 Interface driver ("GPI.SPS") must be used. This driver and its set of special commands are described in Section 6. When evaluated, the expression for an interface number must be between Ø and 3, inclusive.

The string expression following the keyword **HAS** is a driver-dependent interrupt specification. It must be a string accepted by the driver for the given ILUN (or interface). If the HAS and expression are omitted, the driver's default string is used. If the driver has no default string, an error is issued.

The expression following the keyword **AT** is the priority level for the interrupt routine. It must evaluate to an integer between Ø and 126, inclusive. If the AT and expression are omitted, a driver-dependent default value is assumed.

The expression following the keywords **AS TASK** is the task number for the interrupt routine. It must evaluate to an integer between Ø and 126, inclusive. If the AS TASK and task number are omitted, the number of the currently executing task is used. However, if the WHEN is entered in immediate mode and the AS TASK and task number are omitted, task number Ø is assigned to the interrupt routine.

The keyword **GOSUB** precedes the line number of the user-written interrupt routine -- the subroutine to which control is to be passed.

Application Example:

Below is a simple example of using the WHEN command. After the instrument driver is LOADed and the instrument is ATTACHed, the WHEN statement (line 110) allows the program to recognize the pushing of the DPO call button as an interrupt. Until the WHEN is executed, you could push the call button as much as you like, but the program would not respond. After line 110 executes, however, pushing Call Button 1 causes program control to transfer to the interrupt routine that GETs data from the instrument. Because WA is declared as a waveform (line 50), all four parts of the waveform -- the array, data sampling interval, horizontal units and vertical units -- are defined when the data is acquired by line 1040. When the RETURN is encountered, the subroutine terminates and control returns to the calling program.

```
10 REM APPLICATION OF THE WHEN COMMAND
20 REM USING THE DPO
30 REM LOAD THE DRIVER AND
40 REM ATTACH THE INSTRUMENT
50 LOAD "DPO.SPS"
60 ATTACH #1 AS DPO0:
70 REM DECLARE A WAVEFORM
80 WAVEFORM WA IS AA(511),SA,HA$,VA$
90 REM ENABLE CALL BUTTON 1 AS AN INTERRUPT
100 REM WITH A PRIORITY HIGHER THAN DEFAULT
110 WHEN #1 HAS "CB1" AT 60 GOSUB 1020
120 REM PROGRAM CONTINUES
 .
 .
 .
900 RETURN
1000 REM CB1 INTERRUPT SUBROUTINE
1010 REM STORE SIGNAL IN DPO'S MEMORY LOCATION A
1020 PUT "STO" INTO #1,"A"
1030 REM ACQUIRE THE WAVEFORM
1040 GET WA FROM #1,"A"
1050 REM REST OF INTERRUPT ROUTINE
 .
 .
 .
1500 RETURN
```

WRITE (Nonresident)

Examples:

```
500 WRITE #N,WA
150 WRITE #1,A(5:50,10),C$
200 WRITE #X-6,A(I),D2$
250 WRITE #4,"THIS IS THE END"
```

Syntax Form:

$$[\text{line no.}]\ \textbf{WRITE}\ \textbf{\#}\text{expression},\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\\ \text{string expression}\end{array}\right\}\left[,\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\\ \text{string expression}\end{array}\right\}\right]\ldots$$

Descriptive Form:

$$[\text{line no.}]\ \textbf{WRITE}\ \textbf{\#}\text{target plun},\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\\ \text{string expression}\end{array}\right\}\left[,\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\\ \text{string expression}\end{array}\right\}\right]\ldots$$

Purpose:

To store floating-point or integer values or ASCII characters on a peripheral device.

Discussion:

The WRITE command outputs numeric data in a binary format to the specified peripheral for later input by a READ statement. Since the binary value of numeric data is written (as opposed to the ASCII representation of numbers output by the PRINT statement), less peripheral storage space is used. Strings, however, are still output in ASCII format. The READ/WRITE pair is provided for ease of inputting/outputting arrays and waveforms.

The WRITE command accesses a file or device by its peripheral logical unit number (PLUN), not by name. Before you can WRITE to a file or device, you must OPEN it FOR WRITE thereby assigning the PLUN. If the peripheral is OPENed FOR READ or UPDATE, a fatal error results.

If an array or waveform is being output by WRITE when a Control-P is typed at the terminal, the entire array or waveform is output before the program terminates.

[When the WRITE command outputs values to a file, it also writes into the file data descriptors that describe the type and size of the data. The data descriptors are later used by READ when inputting the data. These data descriptors need not concern a BASIC user unless the file being output will be accessed by software other than TEK SPS BASIC. (The TEK SPS BASIC data descriptors are described in Appendix E.) Because the WRITE command writes data descriptors on a file and stores numbers in binary format, not ASCII, a file output by WRITE and input by READ is sometimes called a formatted binary file.]

Using the Syntax Options:

The expression following the pound sign (#) is the peripheral logical unit number (PLUN) to which the data is output. The expression, when evaluated and rounded to an integer, must between 1 and n, where n is the number of PLUNs allowed at system initialization time (default of six). The terminal keyboard, PLUN zero, may not be specified.

The list of data to be output may include expressions, array expressions, waveform expressions, and string expressions.

WRITEU (Nonresident)

Examples:

```
15Ø WRITEU #3,A,INTEGER B,C
25Ø WRITEU #2<9>,A,A$=1Ø
35Ø WRITEU #F,A(11:2Ø),X$=LEN(X$),Y
45Ø WRITEU #N<M>,N+1,A(1:5)/2,Y$=N
```

Syntax Form:

```
                                                  ( [INTEGER] { expression       } )
[line no.] WRITEU #expression[<expression>],     <            { array expression }  >
                                                  ( string expression = expression )

                       [   ( [INTEGER] { expression       } ) ]
                       [ , <           { array expression }  > ]...
                       [   ( string expression = expression ) ]
```

Descriptive Form:

```
[line no.] WRITEU #target plun [<record number>],
                         ( [INTEGER conversion flag] { expression       }    )
                         (                           { array expression }    )
                         ( string expression = number of characters in string )

                       [   ( [INTEGER conversion flag] { expression       }    ) ]
                       [ , (                           { array expression }    ) ]...
                       [   ( string expression = number of characters in string ) ]
```

Purpose:

To output program data to a DEC RT-11 FORTRAN-compatible data file (a file without TEK SPS BASIC data descriptors) or to a record I/O file (a TEK SPS BASIC random-access file).

Discussion:

Data output to a peripheral by the WRITEU command can be input from the peripheral by a DEC RT-11 FORTRAN program or by the READU command.

A numeric expression or an array expression is output in two-word, single precision, floating-point format <u>unless</u> it is immediately preceded by the keyword INTEGER. When the keyword INTEGER is used, the result of the expression (or each element of an array expression) associated with the INTEGER keyword is truncated to an integer and output in one-word (16-bit) integer format. Waveforms are not output by the WRITEU command.

Strings are output as one byte per ASCII character. The expression following the equal sign specifies the number of characters to write. The string output is fitted to this length. If the length of the string is less than this value, trailing blanks (spaces) are added to the string to make it the specified length. If the string is longer than the given value, only the specified number of characters is output.

The byte count required for the different expressions are:

	By default:	**Preceded by INTEGER:**
Expression	4 bytes	2 bytes
Array expression	4 bytes/element	2 bytes/element
String expression	number of bytes specified in expression following equal sign (=)	error

TEK SPS BASIC does not perform integer arithmetic; the results of all numeric expressions are floating-point values. Thus, how WRITEU stores numeric values in a file cannot depend on the source elements of the expression. The output format depends only on the presence or absence of the keyword INTEGER. The following example shows how the WRITEU command outputs various expressions:

```
100 INTEGER I(2)
110 DIM A(2)
 .
 .
 .
500 OPEN #N AS F$ FOR WRITE
510 WRITEU #N,INTEGER A,C$=10,I,X
```

Since the first array expression in line 51Ø (A) is preceded by the INTEGER keyword, the results are written to the file in integer format. The second array expression (I) is not preceded by the keyword INTEGER, so the results are written in floating-point format. That A is a floating-point array or I is an integer array is of no consequence. Therefore, the data will be output as follows:

A(Ø)	output as first 2 bytes (truncated to integer)
A(1)	output as next 2 bytes
A(2)	output as next 2 bytes
C$	output as next 1Ø bytes (string of 1Ø characters)
I(Ø)	output as next 4 bytes (converted to floating point)
I(1)	output as next 4 bytes
I(2)	output as next 4 bytes
X	output as next 4 bytes

Altogether, the WRITEU statement in line 51Ø outputs 32 bytes of data.

[WRITEU does not output the TEK SPS BASIC data descriptors as the WRITE command does. Nor does WRITEU output any delimiters between data items, such as a carriage return, the way the PRINT command does. For this
reo ei utt RIUrotecl ormatted binary
files.]

With the regular form of WRITEU, the data is stored serially, starting at the beginning of a file with the first WRITEU statement. Subsequent outputs to the same file continue writing where the previous WRITEU ended.

When the record I/O (input/output) form of WRITEU is indicated -- by the presence of the angle brackets (<>) -- the mode of access is random. Any data record of the file may be written to, in any order. Multiplying the given record number by the data record length determines where on the file the data is written. The length of the data record is computed by summing the byte count of the items in the output list of the WRITEU statement. The byte count required for each data type is discussed above.

The record length is calculated each time a record I/O form of WRITEU is executed. Thus, when using record I/O, you must output an entire data record with each WRITEU statement, even if you want to write only a part of a record.

The WRITEU command accesses a file or device by its peripheral logical unit number (PLUN), not by name. Before executing WRITEU, the peripheral must be OPEN FOR WRITE or UPDATE, depending on which form of WRITEU is used. When the regular (sequential-access) form of WRITEU is used, the peripheral must be OPEN FOR WRITE. To use the record I/O (random-access) form of WRITEU the file must be OPEN FOR UPDATE. Record I/O files can only be stored on directory-structured devices.

Using the Syntax Options:

The expression following the pound sign (#) is the peripheral logical unit number (PLUN) to which the data is output. The expression, when evaluated and rounded to an integer, must be between 1 and n, where n is the number of PLUNs allowed at system initialization time (default of six). The terminal keyboard, PLUN zero, may not be specified.

The optional expression in angle brackets (<>) specifies the record I/O form of WRITEU. The expression, when evaluated and rounded to an integer, is used as the number of the data record to be written. **The records are numbered from zero**. When the angle brackets and expression are omitted, the regular form of WRITEU is assumed.

The list of data to be output may include expressions, array expressions, and string expressions but not waveforms. By default, all numeric values (including integer array elements) are written in floating-point format. Any numeric item in the list may be output in integer format by specifying the keyword **INTEGER** immediately preceding the item. A string expression must be followed by an equal sign (=) and an expression indicating the number of characters (bytes) to output.

Application Example:

The description of the READU command has an example of how to use READU. There, two files are used to store data about a group of people -- a name file and an information file. The name file, which is kept in alphabetical order, contains, with each name, the record number of the information file where the rest of the data is stored. To get or change the data in the information file, the name is searched for in the name file. When the name is found, the number associated with the name tells you which record in the information file to read or update.

Here, we want to show how to update a portion of the information file -- the address which is stored as two strings in the first 40 bytes of the file. We assume that the routine to do this calls the binary search subroutine (explained in the READU description) to find the desired record number, R, in the information file (line 3Ø6Ø). If the name asked for in line 3Ø3Ø cannot be found in the name file, the search subroutine returns a negative record number. In this case the address-changing subroutine terminates (line 3Ø8Ø). Otherwise the desired record is read, updated, and rewritten on the file in response to input from the keyboard.

```
297Ø REM SUBROUTINE TO UPDATE ADDRESS
298Ø REM
299Ø REM OPEN INFORMATION FILE FOR UPDATE
3ØØØ OPEN #2 AS DX1:"INFOR.FIL" FOR UPDATE
3Ø1Ø REM GET NAME TO SEARCH FOR
3Ø2Ø PRINT "WHOSE ADDRESS NEEDS CHANGING";
3Ø3Ø INPUT S$
3Ø4Ø REM BINARY SEARCH OF NAME FILE FINDS
3Ø5Ø REM NUMBER OF DESIRED RECORD IN INFORMATION FILE
3Ø6Ø GOSUB 1ØØØ
3Ø7Ø REM EXIT SUBROUTINE IF NAME NOT IN FILE
3Ø8Ø IF R<Ø THEN 319Ø
3Ø9Ø REM READ RECORD THAT NEEDS UPDATING
31ØØ READU #2<R>,F$=1ØØ
311Ø REM GET NEW ADDRESS
312Ø PRINT "ENTER NEW ADDRESS IN 2 LINES"
313Ø PRINT "FIRST LINE:";
314Ø INPUT L1$
315Ø PRINT "SECOND LINE:";
316Ø INPUT L2$
317Ø REM WRITE UPDATED RECORD
318Ø WRITEU #2<R>,L1$=2Ø,L2$=2Ø,SEG(F$,41,1ØØ)=6Ø
319Ø CLOSE #2
32ØØ RETURN
```

Line 31ØØ reads the entire record as a single 1ØØ character string. Then the new address is asked for -- to be entered as two lines in separate strings, L1$ and L2$ (lines 312Ø to 316Ø). Finally, the updated record is replaced on the file in line 318Ø. If either L1$ or L2$ is less than or greater than 2Ø characters in length, it is padded with spaces or truncated to exactly 2Ø characters. The remaining 6Ø bytes of information in the file are rewritten unchanged.

The subroutine OPENs and CLOSEs the information file. If several changes were to be made, normally the subroutine would not do this.

ZERO (Nonresident)

Examples:

```
150 ZERO CT1:"OLD.FIL"
    ZERO DK2:
    ZERO DK:10
160 ZERO CT:/F,A$
```

Syntax Form:

```
                                            ┌expression                        ┐
                                            │┌   ┌F┐    ┐                      │
[line no.] ZERO device name[constant]:      ││ / │ │[,] │ [string expression]  │
                                            │└   └R┘    ┘                      │
                                            └                                  ┘
```

Descriptive Form:

```
[line no.] ZERO device name[drive number]:

              ┌number of directory segments                                          ┐
              └[/forward or reverse switch[,]][file name at which to start zeroing tape]┘
```

Purpose:

To initialize a file-structured peripheral, effectively erasing all information on that peripheral.

Discussion:

When a device is ZEROed, all information on that device is logically erased. Though any data on the device is not actually replaced by zeroes, it is no longer accessible by BASIC.

With a directory-structured peripheral, the ZERO command initializes the directory, effectively canceling all files on the device.

With a serial-access device, an end-of-medium (e.g., end-of-tape) marker is written at the beginning of the medium. If a file name is included, the end-of-medium marker is placed at the beginning of the specified file, logically erasing all information on the medium, including and beyond that file.

All files on the specified device must be CLOSEd before ZERO executes.

CAUTION

The ZERO command removes all information from the peripheral device.

Using the Syntax Options:

The device specified must be file-structured. If the named device does not use the system device driver, its driver must be LOADed before ZERO executes. If the drive number is omitted, zero is assumed.

The optional expression may be used with a directory-structured device. It determines the number of segments allocated for the device directory. The expression, when evaluated and rounded to an integer, must be between 1 and 31, inclusive. When this expression is omitted, a dafult value, which is stored in the device driver, is used. (See the Peripheral Drivers manual for the default number of directory segments provided for a particular directory-structured device driver.)

The space allotted for the directory must be large enough to hold the names of all the files to be stored on the device. If most of the files are large (ten blocks or more), the default value may suffice. However, if most of the files are small (about two blocks in length), you may need several times more than the default number of directory segments. For more guidance on how many directory segments to allocate, see the Peripheral Drivers manual.

The optional file name may be used with a serial-access device. If a file name is specified, that file and all files physically following it on the medium are deleted.

[If the device is serial tape and a file name is given, the **/F** or **/R** switch (Forward or Reverse) may be included. The switch specifies the direction of the tape movement when searching for the named file. If the switch is omitted, the tape is rewound before a forward search begins. The search stops when the file is found or an end-of-tape marker is reached. If the device is not a serial-tape device or the file name is omitted, the switch is ignored.]

SECTION 5

FUNCTIONS

In TEK SPS BASIC, a function returns a value (or in some cases, an array of values) that results from the action of the function using the given argument. A function does not change the value of the argument. Of course, if the argument is a variable and it is also assigned the result of the function, it is altered. But this is caused by the action of the assignment, not by the action of the function operation.

A function can only be used as part of an expression within a statement, such as a LET, PRINT, or IF statement. More than one function may appear in a statement. A function may include other functions in its argument -- even itself.

TEK SPS BASIC has three types of functions: numeric functions, array functions, and string functions. The three types are discussed separately in this section.

Numeric Functions

A numeric function performs math operations on specified values, and returns the result to the expression in which it appears. **Depending on the argument, a numeric function returns a single numeric value or an array of numeric values.** When the argument is a numeric expression, one number is returned. When the argument is an array or waveform expression, an array of numbers is returned -- one number for each element in the argument array or the array associated with the argument waveform. The array of values is the result of the function being applied to each of the array (or waveform) elements in turn.

Notice that the result of a numeric function is never a waveform. When a waveform is the argument, the function operates with its associated array only. The data sampling interval (DSI) and the units strings are not used by the function and are not associated with the result. The waveform argument, as with any function argument, is not altered by the function operation unless it is also assigned the result of the function expression.

Thus, if W1 and W2 are waveforms, the statement:

```
W2 = ABS(W1)
```

will make W2's DSI equal to zero and W2's horizontal and vertical units equal to null--even if they were previously defined. Of course, W1's DSI and unit are unchanged. But the statement:

```
W1 = ABS(W1)
```

will nullify W1's units and DSI. This is not because of the action of the function but because of the rules of waveform assignments as described in the LET command. When a waveform is assigned an array, the waveform's DSI is set to zero and its units are set to null.

Absolute Value Function ABS

Returns: The absolute value of the argument.

Syntax:

$$\textbf{ABS(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\textbf{)}$$

Examples:

```
150 IF ABS(X-Y)>Z THEN 550
270 A=ABS(A)
```

Comments: The absolute value function always returns a nonnegative value according to these rules:

if argument is:	the ABS function returns:
positive	the argument
negative	the negative of the argument
zero	zero

Uses: In TEK SPS BASIC, it is illegal to raise a negative value or zero to a power. However, you can use the absolute value function to raise a negative value to an integer power. For example, to raise any nonzero number X to the fifth power, you could use:

```
X5=ABS(X)^5*SGN(X)
```

Since the power is an odd integer, the answer has the same sign as the number. The sign function (SGN) is used to give the answer the correct sign.

Arctangent Function
ATN

Returns: The arctangent of the argument.

Syntax:

$$\textbf{ATN(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\text{)}$$

Examples:

```
170 A1=ATN(Y/X)
330 PRINT ATN(E/SQR(1-E*E))
```

Comments: The result of the arctangent function is in the range of $\pm$Pi/2 radians.

Uses: Since the arctangent of 1 equals one fourth Pi (Pi/4), the following is a very accurate way of defining Pi in a program:

```
150 PI=4*ATN(1)
```

Cosine Function
COS

Returns: The cosine of the argument.

Syntax:

$$\textbf{COS(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\text{)}$$

Examples:

```
990 TN=SIN(X)/COS(X)
450 PRINT M*COS(N)
```

Comments: The argument is assumed to be an angle expressed in radians.

Uses: The cosine function and the sine function are used to find the other trigonometric functions.

Exponential Function
EXP

Returns: The value of e raised to the power of the argument.

Syntax:

$$\textbf{EXP(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\text{)}$$

Examples:

```
19Ø SH=(EXP(E)-EXP(-E))/2
73Ø IF EXP(X)=N THEN 49Ø
```

Comments: The mathematical constant, e, is the base for the natural logarithm (approximately 2.71828). The allowable range of the argument is from -88.5 to 88. If the argument is out of range, an error message is issued. If the argument is less than -88.5, zero is returned; if the argument is greater than 88, the largest possible number in TEK SPS BASIC (approximately 1.7Ø141E+38) is returned.

Uses: The EXP function is used to find the hyperbolic functions. For example, line 19Ø in the examples above finds the hyperbolic sine of E.

Integer Part Function
ITP

Returns: The integer part of the argument.

Syntax:

$$\textbf{ITP(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\text{)}$$

Examples:

```
21Ø IF ITP(X)=X THEN 73Ø
39Ø PRINT ITP(A/B)
```

Comments: The result is equal to the sign of the argument times the greatest integer in the absolute value of the argument. For example, if the argument is 5.7, 5 is returned; and if the argument is -19.9, -19 is returned.

Uses: The integer part function discards the fractional portion, truncating a value to an integer. To round a positive number P to an integer use:

```
P=ITP(P+.5)
```

To round a negative number G to an integer use:

```
G=ITP(G-.5)
```

While, if the value X can be either positive or negative, use:

```
X=ITP((ABS(X)+.5)*SGN(X))
```

The integer part function can also be used to test for even or odd integers. For example, the statement:

```
77Ø IF X=ITP(X/2)*2 THEN 1Ø1Ø
```

causes transfer of program control to line 1Ø1Ø if X is an even integer.

Log Function
LOG

Returns: The natural logarithm (log to the base e) of the argument.

Syntax:

$$\textbf{LOG(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\textbf{)}$$

Examples:

```
43Ø N=B^(LOG(P)/LOG(B))
55Ø PRINT LOG(ABS(X))
```

Comments: The argument must be greater than zero. If it is not, a warning error is issued and a zero is returned.

Uses: The natural log function can be used to find the logarithm of a positive value, P, to a positive base, B, by the statement:

```
LG=LOG(P)/LOG(B)
```

Similarly, to convert the ratio of two voltages, E1 and E2, to decibels use:

```
DB=2Ø*LOG(E1/E2)/LOG(1Ø)
```

Random Number Function
RND

Returns: A pseudo-random, floating-point number between zero and one.

Syntax:

$$\textbf{RND(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\textbf{)}$$

Examples:

```
1Ø5 A=1ØØ*RND(A)
67Ø IF RND(Ø)<.5 THEN 475
```

Comments: The argument is a dummy argument; it is not used by the function except to determine how many numbers to return. All numeric functions return an array of values when the argument is an array expression or a waveform expression. When this is the case, the random function returns an array with a different random number in each element. But if A is an array, the statement:

```
A=RND(Ø)
```

will set each element in A equal to the same random number.

The pseudo-random number is produced by the random number generator. The number that is returned depends on the seed value used by the random number generator. To change or obtain the current seed value, use the RANDOM command.

Uses: You can use the random function to produce a random number between two value, N1 and N2. Assuming that N1 is less than N2, you could use:

```
RN=(N2-N1)*RND(Ø)+N1
```

Notice that a zero was used as the argument. Since it is a dummy argument, any numeric expression could be used.

Sign Function
SGN

Returns: One of the three values (+1, -1, or 0) to indicate the sign of the argument.

Syntax:

$$\textbf{SGN(}\left\{\begin{array}{l}\text{expression}\\\text{array expression}\\\text{waveform expression}\end{array}\right\}\textbf{)}$$

Examples:

```
440 IF SGN(X)=0 THEN 590
120 X=X*SGN(X)
```

Comments: The sign function returns a value indicating the sign of the argument as follows:

if argument is:	value returned:
positive	+1
negative	-1
zero	0

Uses: The sign function can be used to execute different portions of a program depending on whether a value is positive, negative, or zero. For example, the statement:

```
100 GOSUB SGN(N)+2 OF 1000,2000,3000
```

would cause a branch to one of three subroutines depending on the sign of N.

Sine Function
SIN

Returns: The sine of the argument.

Syntax:

$$\textbf{SIN(}\left\{\begin{array}{l}\text{expression}\\\text{array expression}\\\text{waveform expression}\end{array}\right\}\textbf{)}$$

Examples:

```
450 CT=COS(X)/SIN(X)
380 PRINT M*SIN(N)
```

Comments: The argument is assumed to be an angle expressed in radians.

Uses: Sometimes, to test a program, you might need a sine wave with a particular amplitude and number of cycles. The following program segment generates a sine wave of N cycles with amplitude A in array B.

```
1ØØ FOR I=Ø TO SIZ(B)-1
11Ø B(I)=A*SIN(N*6.283185/SIZ(B)*I)
12Ø NEXT I
```

Here, 6.283185 is approximately 2 times Pi radians. The array size function (SIZ) returns the number of elements in the array B. Of course, the larger the array, the better the resolution will be.

Square Root Function
SQR

Returns: The square root of the argument.

Syntax:

$$\textbf{SQR(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\textbf{)}$$

Examples:

```
23Ø M=SQR(X*X+Y*Y)
92Ø PRINT SQR(ABS(A*B/C))
```

Comments: The argument must be greater than or equal to zero. If it is negative, an error is issued and the square root of the absolute value of the argument is returned.

Task Function
TSK

Returns: The task number of the currently executing command.

Syntax:

$$\textbf{TSK(}\left\{\begin{array}{l}\text{expression}\\ \text{array expression}\\ \text{waveform expression}\end{array}\right\}\textbf{)}$$

Examples:

```
14Ø A(TSK(Ø),I)=N
88Ø IF TSK(Ø)=J THEN 1ØØØ
```

Comments: The argument is a dummy argument; it is not used by the function except to determine whether to return a single number or an array with each element equal to the same task number.

The task function returns the task number associated with the statement in which it appears. If this function is used in an immediate mode statement, it always returns a 127, the immediate mode task number.

Array Functions

Array functions are provided in TEK SPS BASIC to compute several of the most common array parameters such as the mean value of the array or waveform. Unlike a numeric function, **an array function always return a single value only.** The argument may be the whole array, a zoned portion of the array, or a waveform. The specified array or waveform is not altered by any of the array functions. Except for the CRS function, an array or waveform variable name is the only argument required for an array function.

Cross Function
CRS

Returns: The interpolated position of the first point at which an element of the argument array or waveform equals or crosses the specified threshold value. If the threshold is not equaled or crossed, a minus one (-1) is returned.

Syntax:

$$\textbf{CRS(}\left\{\begin{array}{l}\text{array}\\ \text{waveform}\end{array}\right\}\text{[(expression[:expression])],expression}\textbf{)}$$

Descriptive Form:

$$\textbf{CRS(}\left\{\begin{array}{l}\text{array}\\ \text{waveform}\end{array}\right\}\text{[(starting position[:ending position])],threshold level}\textbf{)}$$

Examples:

```
1ØØ X=CRS(WA(I+3:J+4),.7)
15Ø PRINT (CRS(A,MAX(A))
```

Comments: The cross function differs from other TEK SPS BASIC functions in that it can take on several forms as the syntax indicates.

The purpose of the CRS function is to find the position (the array subscript for one dimensional arrays) at which a specific level is crossed. The search can start at any point in the array. If no location is specified (no starting position specified), the search starts at the first element, element zero.

The array or waveform is searched linearly from the given starting position or from the default starting position of zero. If the element in the starting position is less than the threshold value, the search stops when an element equal to or greater than the threshold is found. If the starting element is greater than the threshold, the search stops when an element equal to or less than the threshold is found.

Searching continues until the threshold is equaled or crossed or the specified ending position is reached. If no ending position is given, the end of the array or waveform is the default ending position.

When the threshold equals the value of an element, an exact position is returned. But when the threshold value is crossed between two elements, an interpolated position is returned. If the threshold is never crossed or equaled, a minus one (-1) is returned.

If the array or the waveform's array is one-dimensional, the position corresponds to a subscript of an element or to an interpolated subscript indicating a position between two elements.

[When the array is two-dimensional, the meaning of the position in terms of the elements' subscripts is obscure. It is still the interpolated position between two elements. But in this case, the integer portion of the number returned is equal to the first element's first subscript times a quantity one greater than the array's maximum second subscript, plus the element's second subscript. So, for a cross position between elements A(I,J) and A(I,J+1) of an array DIMensioned M by N, the integer portion of the returned value equals I*(N+1)+J.]

The CRS function is best understood with the help of some examples. Consider the statement:

```
C1=CRS(W,.5)
```

When this statement is executed, array (or waveform) W is searched. The value of the starting position (element zero in this case, since no starting position was specified), is noted and a search is made for the first array element that crosses the Ø.5 level. If the value of the starting element is less than the threshold of Ø.5 level, the search stops when an element equal to or greater than Ø.5 is found. If the value of the starting element is greater than the threshold level, the search stops when an element equal to or less than Ø.5 is found.

If an element is found that exactly equals 0.5, its position (the array subscript for one-dimensional arrays) is returned. If the threshold is not matched, but is crossed between two consecutive array elements, an interpolated position (an interpolated subscript value) is returned. If the level is never crossed, the function returns minus one (-1).

In this example:

```
X=CRS(W(45),.5)
```

the starting position is specified. The search starts at element 45, and continues until the 0.5 level is crossed, or the end of the array is reached. While in this example:

```
X=CRS(W(45:522),.5)
```

both the starting and ending positions are specified. Here the search starts at subscript 45 of array (or waveform) W and continues until either the array crosses the 0.5 level or element 522 is reached. Note that a waveform may be referenced, even if zoning is used.

Figure 5-1 graphically demonstrates the CRS function. The statement used in the demonstration is:

```
LET X=CRS(A,.02)
```

Referring to the figure, notice that the 0.02 level is crossed between the fourth and fifth array elements (subscripts three and four). The CRS function automatically interpolates the actual crossing point (3.5) and returns that value.

Had the demonstration statement been:

```
LET X=CRS(A(4:8),.02)
```

the result, X, would have been six. This is because only elements four through eight are searched. In this zone of the array, the 0.02 level is crossed at subscript six.

Uses: The CRS function can be combined with other functions to locate a particular element in an array. For example, to find the subscript, MX, of the maximum value in array BX, use the statement:

```
MX=CRS(BX,MAX(BX))
```

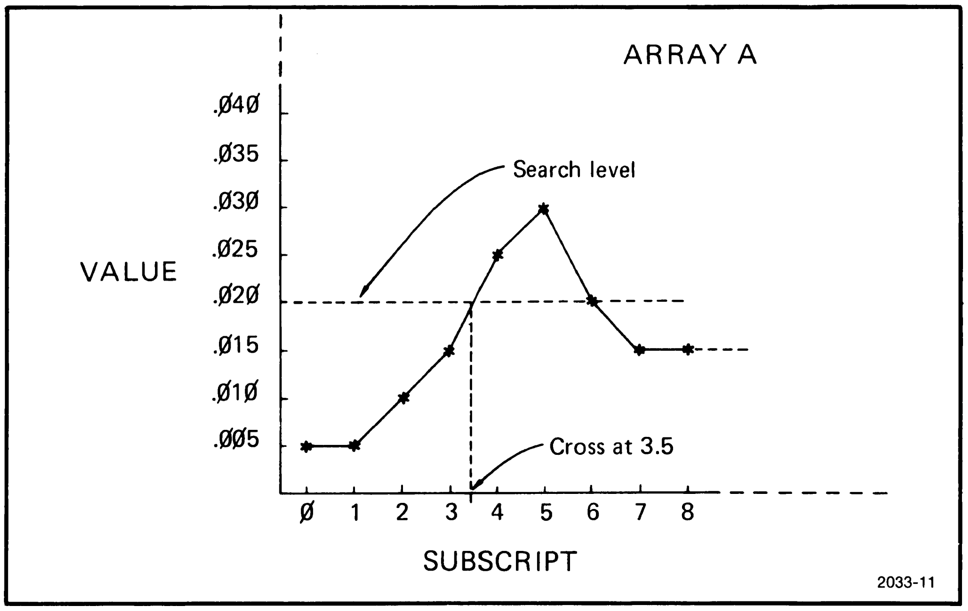


Fig. 5-1. Searching array A with LET X=CRS(A,.Ø2).

Similarly, to find the subscript, MN, of the minimum value in array BX, use:

```
MN=CRS(BX,MIN(BX))
```

Maximum Function
MAX

Returns: The largest value in the argument.

Syntax:

$$\textbf{MAX(} \left\{ \begin{array}{l} \text{array} \\ \text{waveform} \end{array} \right\} \textbf{)}$$

Examples:

```
5ØØ PRINT MAX(AA)
25Ø LET C=MAX(A(5Ø:99))
```

Uses: An example using the MAX and MIN functions together can be seen in the following statement. This single statement will normalize an array to values between zero and one inclusive, an extremely valuable procedure in waveform processing.

```
A=(A-(MIN(A)))/(MAX(A)-MIN(A))
```

The array A is normalized by subtracting the minimum value of the array from each element in array A. Each of the resulting values is then divided by the difference of the maximum and minimum values of the original array. The result, a normalized array, is placed in the original array location A.

Mean Function
MEA

Returns: The mean value of the argument.

Syntax:

$$\textbf{MEA(} \left\{ \begin{array}{l} \text{array} \\ \text{waveform} \end{array} \right\} \text{)}$$

Examples:

```
19Ø IF MEA(A)=N THEN 55Ø
24Ø L=MEA(B(X:Y))
```

Comments: The MEA function sums all the elements in the argument, then divides the sum by the number of elements in the argument.

Uses: One use of the MEA function is to remove a DC level in a waveform. To do this, just subtract the mean value from all elements of the array. For example:

```
W=W-MEA(W)
```

After execution, array W has a mean value near zero.

Minimum Function
MIN

Returns: The smallest value in the argument.

Syntax:

$$\textbf{MIN(} \left\{ \begin{array}{l} \text{array} \\ \text{waveform} \end{array} \right\} \text{)}$$

Examples:

```
99Ø PRINT MIN(B(1ØØ:199))
31Ø IF MIN(A)<MIN(C) THEN 67Ø
```

Uses: Here the MIN function is combined with other array functions to make a fast number sorting routine. The numbers in array A are sorted into a new array B, in ascending order. The contents of array A, however, are destroyed by the process.

```
100 DIM B(SIZ(A)-1)
110 MX=MAX(A)
120 FOR I=0 TO SIZ(A)-1
130 B(I)=MIN(A)
140 A(CRS(A,MIN(A))=MX
150 NEXT I
```

The routine uses a loop to put the current minimum from A into B (line 130). Then in line 140, the current minimum is replaced by A's maximum value, MX. This prevents that element from being the minimum value the next time through the loop. When the routine is finished, B holds the sorted numbers but A is filled with the same number, its maximum value.

This sort can be easily modified to sort numbers into descending order. Either interchange all occurrences of the MIN and MAX functions or change line 120 to:

```
120 FOR I=SIZ(A)-1 TO 0 STEP -1
```

so the array B fills in reverse order.

Root-Mean-Square Function
RMS

Returns: The root-mean-square of the argument.

Syntax:

$$\textbf{RMS(} \left\{ \begin{matrix} \text{array} \\ \text{waveform} \end{matrix} \right\} \textbf{)}$$

Examples:

```
940 X=RMS(A(0:N))
350 PRINT RMS(B)
```

Comments: The root-mean-square is found by summing the squares of the argument's elements, dividing this sum by the number of elements in the argument, and taking the square root of the resulting quotient.

As an example of the RMS function, suppose that one cycle of a sine wave resides in array C. The following program would print 0.7071 on the terminal:

```
100 C=(C-MIN(C))/(MAX(C)-MIN(C))*2-1
110 PRINT RMS(C)
```

In the example, line 1ØØ normalizes the waveform to values in the range of minus one to plus one. Line 11Ø then calculates the RMS value of the waveform and prints the result.

Size Function
SIZ

Returns: The number of elements in the argument.

Syntax:

$$\textbf{SIZ(} \left\{ \begin{array}{l} \text{array} \\ \text{waveform} \end{array} \right\} \textbf{)}$$

Examples:

```
200 FOR I=0 TO SIZ(A)-1
440 IF SIZ(B)<>SIZ(D) THEN 590
```

Comments: The SIZ function always returns the number of elements in the array, not the value to which the array was dimensioned. For example, an array dimensioned to 511 would have a size of 512 and a two dimensional array dimensioned to 3,4 has a size of 2Ø.

Uses: This function is useful when data is fetched from an instrument or peripheral and the destination array is auto-dimensioned. Some instruments capture variable-sized arrays. The SIZ function provides a convenient way of determining the length of the array without tedious programming steps. Thus to dimension an array B to the same size as an auto-dimensioned array A, use:

```
DIM B(SIZ(A)-1)
```

String Functions

TEK SPS BASIC includes a set of string functions which perform operations such as finding a particular sequence of characters in a string or converting strings to numbers or vice versa. **Five of the string functions (CAN, CHR, SEG, STR, and TRM) return a string; the other four (ASC, LEN, POS, and VAL) return a decimal value.**

String functions are always part of a statement, such as a LET or PRINT statement. They can be nested together to produce many useful functions in a single command.

String functions can be deleted at load time (system software initialization) by creating the proper system parameter file with the SYSBLD command. This reduces the size of Resident BASIC, but then the string functions may not be used.

ASCII Function
ASC

Returns: The decimal equivalent of the seven bit ASCII code of the first character of the string argument.

Syntax:

ASC(string expression**)**

Examples:

```
130 LET K=ASC("D")
150 GOTO ASC(N$)-64 OF 100,250,470
```

Comments: The ASC function looks at only one character. If a string of several characters is operated on, the value of only the first character is determined. If the argument is a null string, zero is returned. The ASCII values for all possible characters can be found in Appendix A.

The ASCII function (ASC) is the inverse of the character function (CHR). This means that these two statements are equivalent.

```
PRINT "A"
PRINT CHR(ASC("A"))
```

The ASCII function examines only the first character in the string argument. However, by using it with the segment function (SEG) you can find the ASCII values of all the characters in a string. In this routine, an array A is dimensioned to have as many elements as there are characters in a string. (The LEN Function returns the length of a string.) Then the array is filled with the ASCII values of the characters in A$. Since the characters in a string are numbered from 1 and array elements, from zero, the ASCII value of the Ith character in the string is put into A(I-1).

```
100 DELETE A
110 DIM A(LEN(A$)-1)
120 FOR I=1 to LEN(A$)
130 A(I-1)=ASC(SEG(A$,I,I))
140 NEXT I
```

If A$ is the string "ABCDEF", after the routine executes, the array A would hold the following values:

```
A(0) : 65          A(3) : 68
A(1) : 66          A(4) : 69
A(2) : 67          A(5) : 70
```

Cancel Function
CAN

Returns: A modified version of the argument string in which matching characters on either side of a slash (/) are cancelled.

Syntax:

CAN(string expression)

Examples:

```
150 B$=CAN(F$)
155 C1$=CAN("VV/V")
```

Comments: The function searches the specified string for a slash (/). If found, each character preceding the slash (/) is compared to every character following the slash (/). If a match is found, the two matching characters are removed from the string. If a match is not found, the original string is returned.

For example, if the specified string contained the characters "AA/VA", the CAN functions would cancel the matching characters and return the string "A/V".

Uses: This function is performed automatically on waveform units during waveform arithmetic. For example, a waveform representing a voltage with vertical units of "V" for volts when multiplied by a waveform representing current with vertical units of "A" for amps produces a product waveform whose vertical units are "VA", or volts times amps. If you then divided that product by the current waveform, the intermediate result would be a waveform with vertical units of "VA/A". TEK SPS BASIC automatically cancels the units to produce the final result of "V". The redundancy has been eliminated. The CAN function is provided for users who want to do their own processing in waveform arithmetic.

Character Function
CHR

Returns: A one-character string whose ASCII code in decimal is equal to the argument.

Syntax:

CHR(expression)

Examples:

```
180 C$=C$&CHR(68)
440 F$="FILE."&CHR(I)
560 PRINT CHR(7);
```

Comments: The result of this function is one ASCII character. The character is determined by the value of the specified expression. If the expression, when evaluated and rounded to an integer, is less than zero or greater than 127, it is treated modulo 128. For a value greater than 127, 128 is subtracted from it until the remainder is between zero and 127. For a value less than zero, 128 is added to it until the result is nonnegative.

In the examples above, line 18Ø appends the letter "D" to the string C$. In line 56Ø, the bell character is sent to the terminal, causing the terminal bell to ring.

The character function (CHR) is the inverse of the ASCII function (ASC). Thus, these two statements are equivalent:

```
PRINT 65+N
PRINT ASC(CHR(65))+N
```

A complete list of all ASCII characters and their decimal values is included in Appendix A.

Uses: Some control characters cannot be entered as a string directly from the keyboard, as they are used as editing commands. These control characters and their decimal equivalents are:

Carriage return	=	13
Control-P	=	16
Control-U	=	21
Rubout	=	127

The CHR function allows these characters to be entered indirectly. For example, since 13 is the decimal value of the ASCII code for a carriage return, the statement:

```
PRINT "XXXXXXX";CHR(13);"ZZZZZZZ"
```

would print seven Z's on top of seven X's.

Length Function
LEN

Returns: The number of characters in the specified string.

Syntax:

LEN(string expression)

Examples:

```
400 X=LEN("HELLO"&CC$(8))
200 FOR N=1 TO LEN(T$)
```

Comments: The total number of characters in the string, including nonprinting control characters and any leading or trailing blanks, is found by this function.

Position Function
POS

Returns: The position of the first occurrence of the second string within the first string.

Syntax:

POS(string expression,string expression,expression)

Descriptive Form:

```
POS(source string,search string,starting position)
```

Examples:

```
150 IF POS(A$,C$(R),1)<>0 THEN 490
345 K=POS(D$,"START",X)
```

Comments: This function finds the location of one or more characters in a string. The first string specified is the string searched. The second string is the sequence of characters to look for. The last argument specified tells the POS function where to start the search. The positions are numbered from left to right, starting with 1.

If the second string is not found in the first string, the function returns a zero. If the second string is a null string (a string with length zero), then the value of the third argument is returned.

If the starting position (third argument) is less than 1, the first position is used. If it is greater than the length of the first string, zero is returned.

As an example, suppose A$ contains ABCDEFGHIJKLM, and B$ contains GHI. The statement

```
PRINT POS(A$,B$,1)
```

would cause the number seven to be printed on the terminal.

Uses: The position function can be used to find a substring in a string or to test for the presence of a substring in a string. Besides this, POS can be used to find the delimiters that divide a string into substrings.

For example, this routine uses POS, as well as the length (LEN) and segment (SEG) functions to separate A$ into its three substrings. Here the substrings are of indefinite length but they are delimited by exclamation marks (!). Line 100 finds the position of the first exclamation mark. Line 110 finds the position of the second by making the starting position of the search one greater than the position of the first exclamation mark. Then lines 120 through 140 use this information to break A$ into three separate strings.

```
100 L=POS(A$,"!",1)
110 M=POS(A$,"!",L+1)
120 T1$=SEG(A$,1,L-1)
130 T2$=SEG(A$,L+1,M-1)
140 T3$=SEG(A$,M+1,LEN(A$))
```

If A$ is equal to "BUY NOW!PAY LATER!NOTHING DOWN", then after the routine executes the three new strings would contain the following.

```
T1$: BUY NOW
T2$: PAY LATER
T3$: NOTHING DOWN
```

Segment Function
SEG

Returns: A substring of the specified string.

Syntax:

SEG(string expression,expression,expression**)**

Descriptive Form:

SEG(source string,position of substring's first character,
position of substring's last character**)**

Examples:

```
700 Z$=SEG(A$,K,J)
670 IF SEG(A$,I,I)<>"5" THEN 880
```

Comments: The substring is taken from part of the specified source string. The first expression specifies the position of the first character of the substring. If this expression has a value of one, the new string will begin with the first character of the source string. Likewise, if this value is five, the new string begins with the fifth character in the source string. The second expression specifies the last character in the new string. The positions in the source string are numbered from left to right, starting with 1.

For example, if A$ contained the string "THIS IS A STRING EXAMPLE", which has a length of 24 characters, the statement

```
PRINT SEG(A$,11,16)
```

would print the following string:

```
STRING
```

If the starting position (first expression) has a value less than 1, the first position is used. If the ending position is greater than the length of the source string, the actual length of the source string is assumed. Should the starting position be greater than the length of the string, a null string (a string of no length) is returned. Likewise, if the ending position is less than 1, a null string is returned. The same is true if the ending position is less than the starting position.

Uses: This function is used to create one or more new strings from a source string, or to look at part of a source string. So, while the position function (POS) lets you quickly find a particular substring in a source string, when you need to find one of a range of characters in a source string, use SEG.

For example, this routine looks for a string of integers in the source string A$, and returns it in a new string, N$.

```
100 FOR I=1 TO LEN(A$)
110 IF SEG(A$,I,I)>="Ø" THEN IF SEG(A$,I,I)<="9" THEN 170
120 NEXT I
130 REM NO NUMERS IN STRING
140 N$=""
150 GOTO 220
160 REM NUMBER FOUND IN STRING
170 FOR J=I+1 TO LEN (A$)
180 IF SEG(A$,J,J)<"Ø" THEN 210
190 IF SEG(A$,J,J)>"9" THEN 210
200 NEXT J
210 N$=SEG(A$,I,J-1)
220 RETURN
```

The loop in lines 100 to 120, scans across the source string, one character at a time, looking for a numeric character. Notice that the compound IF statement in line 110 is true only if the character in the Ith position is a character between Ø and 9, inclusive. If a numeric character is found, that loop is exited. The loop variable I points to the position of the first character in the integer substring.

Now another loop is used to find the end of the integer substring. The FOR loop in lines 17Ø to 2ØØ scans for a nonnumeric character. When that is found or the end of the string is reached, the second FOR loop's variable, J, points to the position just to the right of the last numeric character. Thus, line 21Ø sets N$ to the integer substring in A$.

If no numeric character is found by the first loop, N$ is set to the null string in line 14Ø.

String Function
STR

Returns: A string which represents the numeric value of the specified expression.

Syntax:

```
STR(expression)
```

Examples:

```
55Ø PRINT STR(K+45.6)
77Ø V$=A$&STR(J)
```

Comments: The STR function is used to create a string of ASCII characters which represent the decimal value of an expression. It is the opposite of the VAL function. This is the same routine BASIC uses when values are printed at the terminal with the PRINT statement.

The string created by STR will not have any leading spaces or zeros, but will have a leading minus sign (-) if the value of the expression is negative.

Uses: The STR function can be used whenever you need to convert a floating-point value to a string representation of its decimal value. For example, these statements:

```
1ØØ FOR I=1 TO 4
11Ø OPEN #I AS "DATA."&STR(I) FOR READ
12Ø NEXT I
```

open four files named "DATA.1", "DATA.2", "DATA.3" and "DATA.4" for READ on the system device.

Trim Function
TRM

Returns: A copy of the argument string trimmed of any trailing blanks.

Syntax:	**Examples:**
TRM(string expression**)**	6ØØ GF$=TRM(GF$) 7ØØ PRINT TRM(P$(J))&SEG(B$,1,2Ø)

Comments: Strings can be "tidied up" with the TRM function. Any spaces at the end of the specified string are removed by this function. The function starts at the end of the string, and terminates with the first non-space character.

Uses: When a statement like

```
WRITEU #1,A$=4Ø
```

is used to write a string into a data file, blanks are added to the end of A$ if the actual length of A$ is less than the 4Ø characters specified. Later, when A$ is read from the file by READU, the trim function can be used to remove the added blanks. For example:

```
READU #1,A$=4Ø
A$=TRM(A$)
```

Value Function
VAL

Returns: The decimal value represented by the specified string.

Syntax:	**Examples:**
VAL(string expression**)**	589 PRINT VAL (CC$) 7ØØ X=VAL(SEG(A$,1,4))

Comments: The VAL function is the opposite of the STR function. Here, the result is a number equal to the decimal value represented by the string. It is the same routine used by the INPUT statement to convert ASCII characters to their decimal value.

If the string does not represent a number, a warning error is issued and zero is returned. If the number is too large to convert, a warning error is generated and the largest possible number (approximately 1.7Ø141E+38) with correct sign is returned. Likewise, if the number is too small to convert, zero is returned and a warning error is given. The legal characters in a string representation of a decimal number are the digits Ø through 9, plus and minus signs (+, -), the decimal point (one only), and the letter E. Blanks in the string are ignored.

Uses: The VAL function is used to convert a string representation of a decimal number to a floating-point value. For example, if the scale factor of an instrument is returned in all but the first three characters of a string, A$, the statement:

```
SF=VAL(SEG(A$,4,LEN(A$)))
```

defines the scale factor, SF.

SECTION 6

IEEE 488 INTERFACE DRIVER

This section describes the low-level IEEE 488 Interface driver which accompanies the TEK SPS BASIC System Software. This software module, named "GPI.SPS", is an instrument driver for either a CP4100/IEEE 488 Interface or a CP1100/IEEE 488 Interface.

The section starts with a brief discussion of the IEEE 488 Standard-1975 Interface Bus which is also called the General Purpose Interface Bus (GPIB). Then the functional subsets of the IEEE 488 standard supported by the interface driver are presented. Next is a general discussion of how the driver is used, particularly with the instrument commands GET, PUT, WHEN, and IGNORE. The command descriptions for the interface driver's nonresident commands are at the end of the section.

Introduction to the IEEE 488 Bus

The IEEE 488 bus is a versatile instrument bus designed to provide an effective communications link for data and instructions. The bus itself is entirely passive. The active components of the interface are contained within each device. Instruments designed to operate according to this universal standard can be connected directly to the bus and operated by a controller with appropriate programming. The instructions and data generated by instruments can be coded in either ASCII or binary. The IEEE standard specifies only the mechanical, electrical, and functional aspects of the interface. The operational, or device dependent, aspects of the system are purposely not specified to allow greater flexibility as to the types of devices that can be interconnected.

A Typical System

The IEEE 488 bus uses eight data lines and eight control lines. Information is transferred bit-parallel, byte-serial by an asynchronous handshake. The handshake signals guarantee that each data byte has been transferred properly before allowing another byte to be transferred across the bus. This allows instruments with different data transfer rates to operate together if they conform to the handshake state diagrams defined in the IEEE 488 standard.

Types of Instruments. Instruments connected to the bus can be classified as either controllers, talkers, or listeners. A controller designates which devices are to talk or listen and exercises other bus management functions; at any given time, there can be only one controller-in-charge. A talker is a device capable of transmitting data and instructions on the Data lines; there can be only one talker at a time to avoid confusion in message and data transfer. A listener is a device capable of responding to data or instructions received on the Data lines; there can be more than one listener at a time.

A device need not be a talker or listener or controller at all times. It may be idle part of the time. Some devices (such as a digital multimeter) may alternately function as talkers or listeners depending on whether they are generating data or receiving instructions.

A typical system is diagrammed in Fig. 6-1. It includes a controller (such as a TEKTRONIX CP4165 Controller), a talker (such as a counter or digital multimeter), and a listener (such as a line printer or signal generator). Also included is a TEKTRONIX 7912AD Programmable Digitizer which may either talk or listen.

Types of Messages. Messages on the bus are either interface messages or device-dependent messages. Interface messages are used to manage the interface functions of the instruments. They designate talkers and listeners, determine local or remote operation of devices, indicate service requests, and communicate other important interface conditions. Device-dependent messages, by contrast, are not used to change the state or configuration of the interface, but are used to control the operating modes or device functions of designated instruments. Device-dependent messages can also be data, such as waveform data generated by the TEKTRONIX 7912AD Programmable Digitizer.

Maximum Number of Devices. Up to 15 devices can be connected on the IEEE 488 bus. More than 15 devices can be interfaced if they are not directly connected to the bus but are interfaced through another device. Such a scheme is used for the programmable plug-ins housed in a 7000-Series programmable mainframe such as the 7912AD; the mainframe interfaces the programmable plug-ins to the bus. Secondary addresses are used to distinguish the mainframe and the plug-ins. More than half of the main devices connected at any time must be powered-up for the system to be operational.

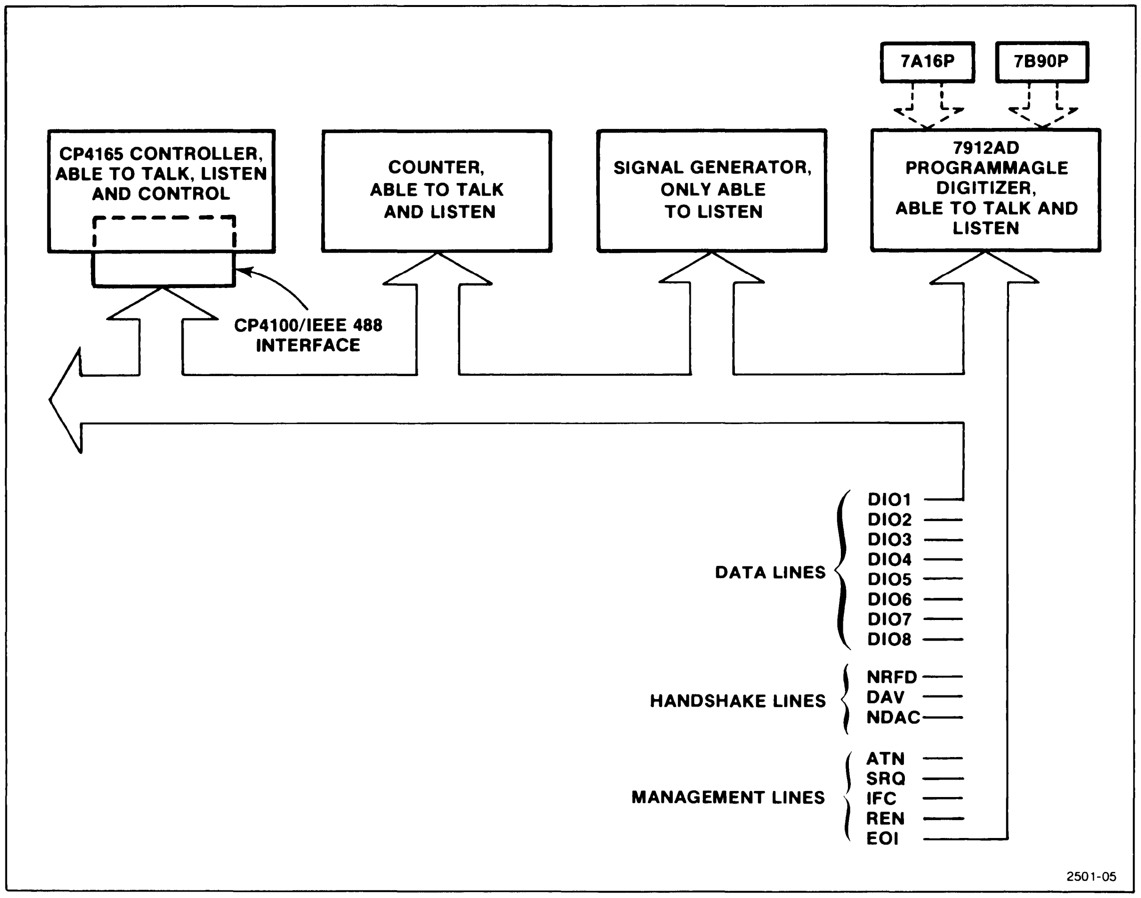

7A16P
7B90P
CP4165 CONTROLLER, ABLE TO TALK, LISTEN AND CONTROL
COUNTER, ABLE TO TALK AND LISTEN
SIGNAL GENERATOR, ONLY ABLE TO LISTEN
7912AD PROGRAMMAGLE DIGITIZER, ABLE TO TALK AND LISTEN
CP4100/IEEE 488 INTERFACE
DATA LINES
DIO1
DIO2
DIO3
DIO4
DIO5
DIO6
DIO7
DIO8
HANDSHAKE LINES
NRFD
DAV
NDAC
MANAGEMENT LINES
ATN
SRQ
IFC
REN
EOI
2501-05

Fig. 6-1. A typical system based on the IEEE 488 Bus.

Maximum Cable Length. The maximum length of cable that can be used to connect a group of devices on the bus is 2 meters times the number of devices or 2Ø meters, whichever is less. Cables may be connected in either a star or a linear configuration, or in a combination of the two methods. (See Fig. 6-2.)

Electrical Specifications. The relationship between the binary logic states of the bus and the voltages present on the signal lines is as follows:

Logical 1 corresponds to a low voltage level (+Ø.8 V or less) and the signal is said to be "asserted".

Logical Ø corresponds to a high voltage level (at least +2.Ø V) and the signal is said to be "unasserted".

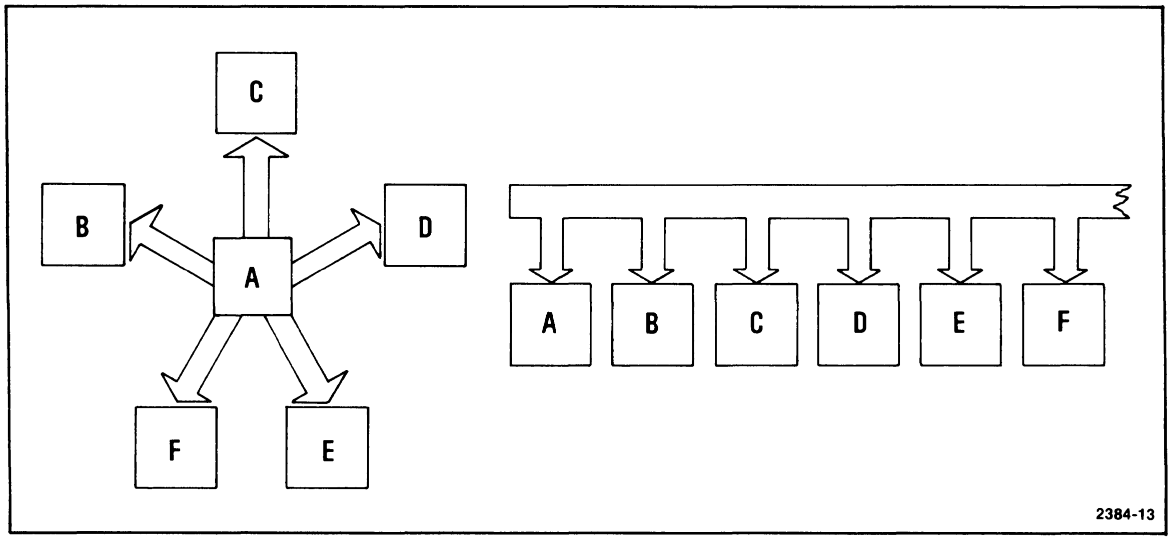


Fig. 6-2. An IEEE 488 system can be configured in either a star or linear manner without impairing the bus electrical characteristics.

The electrical states are based on standard TTL (Transistor-Transistor Logic) levels where the power source does not exceed +5.25 Volts DC referenced to logic ground.

Bus Signal Lines

The IEEE 488 bus is functionally divided into eight data lines and eight control lines. The eight control lines consist of three handshake lines and five management lines. This bus structure is diagrammed in Fig. 6-1.

Data Lines

The eight Data Input/Output lines (DIO1 through DIO8) are bidirectional active-low lines used to convey data or device-dependent messages. Device addresses and universal commands are also transferred over these lines when ATN is asserted. One byte of information is transferred over the bus at a time. DIO1 represents the least significant bit in the byte; DIO8 represents the most significant bit. Data is transferred in byte-serial, bit-parallel fashion. Data bytes can be formatted in ASCII with or without parity, or they can be formatted in machine-dependent binary code. The term "machine-dependent binary code" refers to an internal binary format used by a device to store certain programs and data.

Control Lines

The **three handshake lines** are used to communicate a handshake sequence that is executed between the talker and all designated listeners each time a byte is transferred over the data lines. This handshake sequence prevents the talker from placing a new byte on the bus until the slowest listener has captured the previous byte. Thus the talker can not transmit at a rate faster than can be received by the slowest listener. The three active-low handshake lines are NRFD, DAV, and NDAC. (See Fig. 6-3 for a basic timing relationship among these signals). Their functions are:

NRFD (Not Ready For Data) -- This signal line is asserted until all assigned listeners are ready to receive the next data byte. When all of the assigned listeners have released NRFD, the NRFD signal is unasserted, thereby allowing the talker to place the next byte on the Data lines.

DAV (Data Valid) -- The DAV signal line is asserted by the talker shortly after placing a valid byte on the Data lines. This tells each listener to capture the byte presently on the Data lines. DAV can not be asserted until NRFD has been unasserted.

NDAC (Not Data Accepted) -- This signal line is asserted until all the listeners have captured the byte currently on the data lines and released NDAC. When the slowest listener has captured the data byte and released NDAC, NDAC is unasserted thereby allowing the talker to remove the byte from the data lines. At that point, the DAV line is unasserted and the entire handshake cycle is repeated.

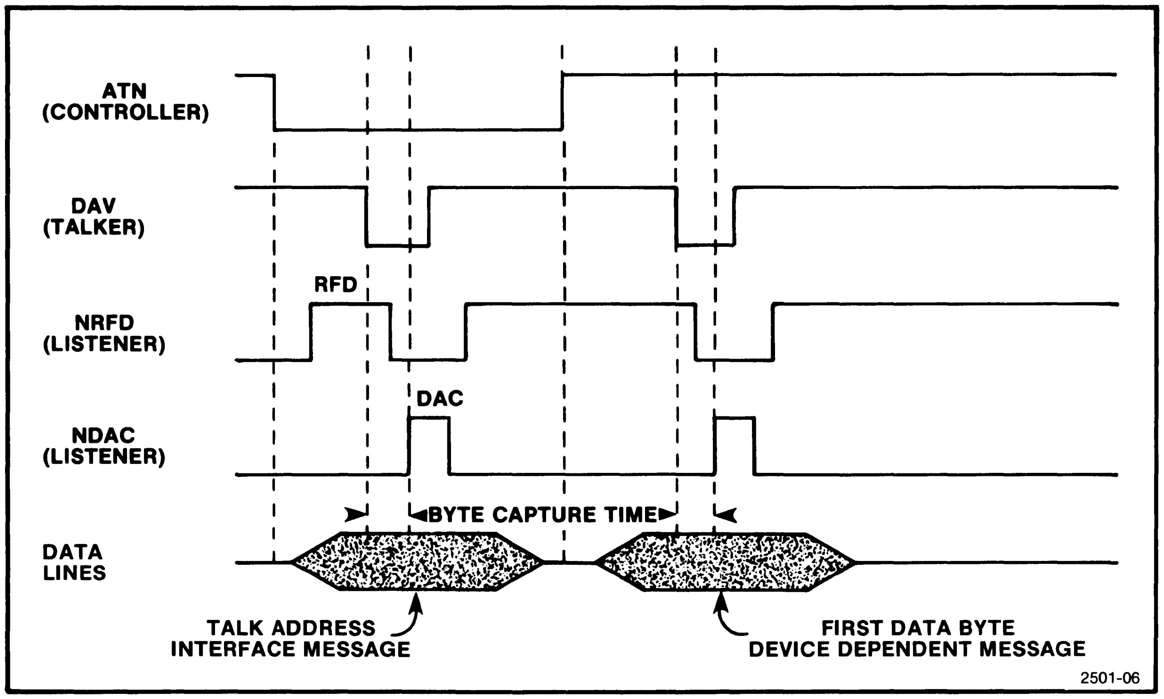


Fig. 6-3. A typical handshake sequence.

The **five management lines** are used to control data transfers over the data lines. The management lines perform important interface operations such as detecting an interrupt from a device, setting a device to remote control, and flagging the end of a message. These five active-low signal lines are ATN, IFC, SRQ, REN, and EOI; their functions are:

ATN (Attention) -- Asserted by the controller-in-charge to specify how information on the data lines is to be interpreted. When ATN is not asserted, the information on the data lines is interpreted as device-dependent messages and data. When ATN is asserted, the data lines convey universal commands, addressed commands, talk addresses (My Talk Address, MTA), listen addresses (My Listen Address, MLA), or secondary addresses (My Secondary Address, MSA). Just which addresses and commands are sent depends upon the byte currently on the data lines. The codes corresponding to various commands and addresses are defined in Appendix E of the IEEE 488 standard.

IFC (Interface Clear) -- Asserted by the system controller to initialize the interface functions of all instruments to an inactive state and return control to the system controller. The IFC function effectively performs an UNListen, an UNTalk and a Serial Poll Disable and resets all devices except the system controller to the idle state.

SRQ (Service Request) -- Asserted by an instrument to request service from the controller-in-charge. The controller usually interrupts its current task and conducts a serial poll to determine which device asserted SRQ. The controller can then branch to an interrupt service routine where appropriate action is taken. After the interrupt has been processed, the controller may resume execution of the previous task.

EOI (End Or Identify) -- Asserted by a talker to indicate the last byte of its message. When EOI is asserted with ATN, the controller is conducting a parallel poll of the devices connected to the bus.

REN (Remote Enable) -- Asserted by the system controller to allow devices on the bus to go to Remote mode, thereby allowing remote control of their programmable functions. When in Remote mode, the front panels of the instruments are disabled except for any non-programmable functions.

Bus Messages

As previously noted, messages on the data lines are either interface messages or device-dependent messages. When the ATN line is asserted by the controller, all devices "pay attention" since interface messages are to be transferred over the data lines. (By "pay attention" it is meant that all devices handshake and process all bytes transferred on the bus.) Interface messages can generally be classified as follows:

1) talk addresses
2) listen addresses
3) secondary addresses
4) universal commands
5) addressed commands

The first three categories refer to how a device is to be addressed. That is, they designate whether a device is to be a talker or a listener. To designate a device as a talker, the controller asserts ATN and places the device talk address on the data lines. Similarly, the controller designates a listener by asserting ATN and placing the device listen address on the data lines. In cases where secondary addressing is designed into a particular device, it is necessary to transmit the device secondary address with ATN asserted following the primary talk or listen address.

The fourth category listed (universal commands) consists of those interface commands which affect all devices connected to the bus, regardless of whether they are currently addressed as talker or listeners. Examples of universal commands are LLO (Local Lockout) and DCL (Device Clear).

The fifth category in the list (addressed commands) consists of those interface commands which affect all devices currently addressed as listeners. One exception, TCT (Take Control) is sent to a single device that has been addressed to talk. Other examples of addressed commands are GTL (Go To Local) and GET (Group Execute Trigger). A complete list of universal and addressed commands is provided in Appendix E of the IEEE 488 standard.

In contrast to interface messages, device-dependent messages are sent with ATN unasserted and are transmitted only between a designated talker and one or more designated listeners. A device-dependent message can be either an instruction (e.g., set the input polarity to normal) or data (e.g., 3.456 volts). The format of instructions and data is entirely up to the device designer. Instructions and data are normally coded in ASCII or binary, but this is not required by the IEEE standard.

This has only been a brief introduction to the IEEE 488 interface. Further information can be found in IEEE Standard 488-1975, **IEEE Standard Digital Interface for Programmable Instrumentation**. A detailed description of the actual handshake timing sequence is covered in Appendix B of the standard.

Introduction to the IEEE 488 Interface Driver

IEEE 488 Interface Function Subsets

The IEEE 488 standard is designed in such a way that not all devices on the bus need to have the same capability to comply with the standard. The instrument designer can choose from a "menu" of ten device functions, and implement only those capabilities (known as "functional subsets") that are appropriate to a particular device. The functional subsets are described in detail in the standard. The degree to which the controller (computer) with a CP4100/IEEE 488 or a CP1100/IEEE 488 interface, operating under TEK SPS BASIC and the low-level IEEE 488 Interface driver (GPI.SPS), implements each of the ten interface functions is described below.

1) **Source Handshake** Function: SH1
The SH function provides a device with the ability to initiate and terminate transfer of messages on the eight data lines. Subset SH1 means it has full capability with no states omitted.

2) **Acceptor Handshake** Function: AH1
The AH function provides a device with the capability to guarantee proper reception of messages on the eight data lines as well as the capability of delaying initiation or termination of such messages. Subset AH1 means it has full capability with no states omitted.

3) **Talker** function: TE5 or T5
The T function enables a device to send device-dependent data (including status information) over the bus to other devices. Subset TE5 means it is an extended talker honoring secondary addresses, responds to serial poll, has talk only mode, and is unaddressed if MLA (My Listen Address) and MSA (My Secondary Address) are received. Subset T5 means it is a basic talker, responds to serial poll, has talk mode only, and is unaddressed if MLA is received. (MLA and MSA are established by software.) Response to serial poll is accomplished by using the driver's nonresident commands plus PUTLOC and GETLOC. However, the driver assumes it is controller-in-charge.

4) **Listener** Function: LE3 or L3
The L function allows a device to receive device-dependent data over the bus from other devices. Subset LE3 means it is an extended listener honoring secondary addresses, has listen only mode, and is unaddressed if MSA (My Secondary Address) is received and it is in the TPAS (Talker Primary Addressed State). Subset L3 means it is a basic listener, has listen only mode, and is unaddressed if MTA (My Talk Address) is received. (MTA and MSA are established by software.)

5) **Service Request** Function: SR1
The SR function enables a device to asynchronously request service from the controller-in-charge of the interface bus. Subset SR1 means it has full capability.

6) **Remote/Local** Function: RL1
The RL function provides a device with the capability to select between two sources of information: remote (programmed control) or local (front-panel control). Subset RL1 means it has full capability.

7) **Parallel Poll** Function: PPØ
The PP function allows a device to present one bit of status to the controller-in-charge without being previously addressed to talk. Subset PPØ means it has no capability for responding to a parallel poll.

8) **Device Clear** Function: DC1
The DC function allows devices to be cleared (initialized) either individually or in groups. Subset DC1 means it has full capability.

9) **Device Trigger** Function: DT1
The DT function allows the operation of a device to be triggered (initiated) either individually or as part of a group. Subset DT1 means it has full capability.

10) **Controller** Function: C1, C2, C3, C4, and C25
The C function provides a device with the capability of sending device addresses, universal commands, and addressed commands over the bus. Subsets C1, C2, C3, C4, and C25 mean it can be system controller, send Interface Clear (IFC) and take charge,

send Remote Enable (REN), respond to a Service Request (SRQ), send interface messages, perform a parallel poll, and take control synchronously. Receiving control and passing control can be implemented through the driver's nonresident commands plus PUTLOC and GETLOC, but these functions are not supported by the driver since it assumes it is controller-in-charge.

Loading the IEEE 488 Interface Driver

In order to use the low-level IEEE 488 Interface driver "GPI.SPS" and its commands, the IEEE 488 capabilities must have been retained when the system software was loaded. This is the default condition. If these capabilities were deleted, however, it means the user-defined parameter file "SYSBLD.DEF" on the system device has the IEEE 488 capabilities parameter set to N for No. In that case, you will have to run SYSBLD to create a new "SYSBLD.DEF" file that retains the IEEE 488 capabilities and then reload the system software. (See the SYSBLD command description in Section 4.)

If IEEE 488 capabilities were retained, the next step is to LOAD the driver into controller memory from the peripheral device. Assuming the driver is stored on the system device, the following command can be used:

```
     LOAD "GPI.SPS"    or simply    LOAD "GPI"
```

where "GPI.SPS" is the name of the IEEE 488 Interface driver module. (The LOAD command assumes the .SPS extension.)

When the driver is no longer needed, it may be removed from memory by executing:

```
     RELEASE "GPI"
```

Notice that again the .SPS extension is assumed.

The IEEE 488 Interface driver supports up to four interfaces. The first interface installed in the system controller is strapped as interface zero. The second interface is strapped as interface one and so on. This allows one controller with four interfaces to control four separate IEEE 488 systems with up to 15 devices (including the controller) connected to each interface.

The low-level IEEE 488 Interface driver "GPI.SPS" is different from other instrument drivers in TEK SPS BASIC. The IEEE 488 interface is not ATTACHed. Instead of associating an instrument logical unit number (ILUN) with the interface, another convention is used. When communicating with the IEEE 488 interface, the specific interface card is indicated by an at sign (@) preceding the interface number. This interface number (Ø,1,2, or 3) is determined by the way the interface is strapped. It is not a logical number assigned by software.

Addressing Instruments on the IEEE 488 Bus

When communicating with instruments on the IEEE 488 bus, it is necessary to address each listener and talker on the bus as well as specifying the interface number.

Each instrument on the bus must have a unique primary address in the range Ø - 3Ø. In addition, an instrument may respond to a secondary address, also in the range Ø - 3Ø. Secondary addresses may be used to distinguish an instrument mainframe from plug-in units, or may be used to specify commands to the instrument.

Instruments are addressed to listen or talk by sending their listen or talk addresses over the bus with the Attention (ATN) line asserted. Talk addresses, listen addresses, and secondary addresses are distinguished as follows: listen addresses consist of the instrument primary address plus 32; talk addresses are the instrument primary address plus 64; and secondary addresses are the instrument secondary address plus 96 (for either talk or listen). For example, sending the sequence

```
33  34  96
```

over the bus with ATN asserted would address two devices as listeners, one with primary address 1 and a second device having primary address 2 and secondary address Ø.

It is important to remember that the proper base (32, 64, or 96) must be added to the primary or secondary address when issuing a command.

Driver Control of the Bus Signal Lines

The IEEE 488 interface standard defines a 16-line bus system. Eight of the lines are used for data transfer, and eight are used for control lines. Although any of the lines can be specifically set or cleared under BASIC program control, the driver software usually controls these lines automatically. The following is a brief discussion of the signal lines and how they are controlled. The commands mentioned are discussed in detail later.

Remote Enable. Once the Remote Enable line (REN) has been asserted, REN remains asserted until the line is explicitly cleared by the SIFLIN command. **REN is not cleared when the driver is RELEASEd.**

End or Identify. End or Identify (EOI) is normally asserted with the last byte of data sent on the bus. The only exceptions are the WASCII and WBYTE commands, which optionally assert EOI. An EOI received from the IEEE 488 bus always terminates a data transfer, even though the expected amount of data has not been transferred or the expected termination character has not been received.

Bus Handshake Lines. The bus handshake lines (DAV, NRFD, and NDAC) are normally under control of the interface hardware and are transparent to the programmer.

Service Request. Service Request (SRQ) is detected by the IEEE 488 interface hardware. TEK SPS BASIC can detect the occurrence of a service request with the WHEN command.

Interface Clear. Interface Clear (IFC) is not normally asserted by the driver software, except when specified by the SIFLIN command.

Attention. The Attention (ATN) line is asserted by the driver software whenever addresses or universal commands are sent. ATN is unasserted whenever the driver is preparing to send or receive messages on the bus.

All of the above bus signal lines (DAV, NRFD, NDAC, ATN, EOI, SRQ, IFC, and REN) may be individually controlled with the SIFLIN command and may be read with TIFL command. These commands are discussed later in this section.

Transferring Data with PUT or GET

Four basic types of variables can be used to accept data in a data transfer by the low-level IEEE 488 Interface driver with a GET command: string variables, integer arrays, floating-point variables, and floating-point arrays. Three basic types of data can be sent by the low-level IEEE 488 Interface driver with a PUT command: numeric expressions, array expressions, and string expressions.

The following conventions characterize data transfers initiated by the Interface driver with the GET and PUT commands.

1. The direct memory access (DMA) data transfer to the controller memory is accomplished by specifying an integer array as the target variable in the GET command. The nonresident command IFDTM (Interface Data Transfer Mode) allows the user to specify the types of DMA transfer desired. IFDTM arguments affecting data transfers are HOG or UNHog, PAcK or UNPack, and High Byte First or Low Byte First. (See IFDTM command description for additional information.) The default data transfer modes of the interface driver are UNP, UNH, and HBF.

2. Data transfers from the IEEE 488 Interface to string variables may be terminated by receiving EOI with a data byte or by receiving the termination character(s) specified by the STERMC command.

3. The default time-out value for the interface driver is five milliseconds. This can be changed by executing the SIFTO (Set InterFace Time-Out) command. Time-out values are approximate and dependent upon controller type.

The GET Command

The GET command is used to fetch data from a specified source (bus-connected instrument or device) and deliver that data to a specified target (variable or array in the controller memory) for storage. The GET command accesses an instrument by asserting ATN (attention) and sending UNT, UNL (untalk, unlisten) to render the bus inactive. Then the controller interface is enabled to listen without ATN asserted. This sequence is followed by a talk address and an optional secondary address. With a communication channel thus established, ATN is unasserted and data from the addressed instrument is read into the specified target until the target argument is satisfied, an EOI occurs, or a preset delay for a transfer has elapsed (time-out). This transaction is followed by another ATN and the UNT, UNL messages. Then NRFD is released and the interface is disabled to listen without ATN asserted. Finally, ATN is unasserted.

The low-level IEEE 488 Interface driver uses this portion of the GET command syntax:

Syntax Form:

```
                  ( variable              ) [  ( variable              ) ]
[line no.] GET   { floating-point array  } [ ,{ floating-point array  } ]...
                 { integer array         } [  { integer array         } ]
                  ( string variable       ) [  ( string variable       ) ]
                 FROM @expression,expression[,expression]
```

Descriptive Form:

```
                  ( target variable              ) [  ( target variable              ) ]
[line no.] GET   { target floating-point array  } [ ,{ target floating-point array  } ]...
                 { target integer array         } [  { target integer array         } ]
                  ( target string variable       ) [  ( target string variable       ) ]
                 FROM @IEEE 488 interface number,
                      primary talk address [,secondary address]
```

The target for the data may be one or more variables, string variables, floating-point arrays, or integer arrays. For an integer array, the data transfer is performed using direct memory access (DMA). The exact amount and format of the data as stored in the target is discussed below.

The expression following the at sign (@) is the interface number of the bus on which the device is connected.

The second expression specifies the primary talk address of the device sending the data. It must be between 64 and 94, inclusive. The optional third expression may be used to specify the secondary address of the device. If used, it must be between 96 and 126, inclusive.

How the data is read and stored depends on the variable specified in the GET command and the current data transfer mode as set by IFDTM. Data is read from the bus and stored in the specified target according to the following table:

Type	Mode	Description
variable:	UNP:	One byte is read from the bus and stored in the variable.
	PAK HBF:	Two bytes are read and stored in the variable. The value stored is the floating-point representation of the first byte times 256, plus the second.
	PAK LBF:	Two bytes are read and stored in the variable. The value stored is the floating-point representation of the second byte times 256, plus the first byte.
	HOG or UNH:	No effect.
floating-point array:	UNP:	One byte is read from the bus and its floating-point representation is stored for each element of the array.
	PAK HBF:	Two bytes are read from the bus and stored for each element of the array. The value stored is the floating-point representatic of the first byte times 256, plus the second byte.
	PAK LBF:	Two bytes are read from the bus and stored for each element of the array. The value stored is the floating-point representatic of the second byte times 256, plus the fir byte.
	HOG or UNH:	No effect.
integer array:	UNP:	One byte is read from the bus and stored for each element of the array.
	PAK HBF:	Two bytes are read from the bus and stored for each element of the array. The value stored is equal to the first byte times 256, plus the second byte.

	PAK LBF:	Two bytes are read from the bus and stored for each element of the array. The value stored is equal to the second byte times 256, plus the first byte.
	UNH:	The transfer is via normal Direct Memory Access (DMA) protocol.
	HOG:	The DMA transfer is accomplished with the interface having exclusive access to the controller bus. No intervention is allowed. Note: For the CP4100/IEEE 488 interface, hog mode is really "burst" mode to allow the processor to refresh the volatile memory.
string variable:		Data bytes are read from the bus and stored in consecutive elements of the string variable specified. Reading is stopped when EOI is detected or the termination character(s) are read. (See STERMC command.) Data transfer modes PAK, UNP, HOG, UNH, HBF, and LBF, set by the IFDTM command, have no effect on transfers into string variables.

Bus Traffic: GET example

Messages	**Comments**
ATN UNT	Controller untalks and unlistens
ATN UNL	all devices.
ATN talk address	Talk address from expression.
[ATN secondary address]	Secondary address from expression.
.	
.	
data from instrument	From talker.
.	
.	
EOI	Data stream terminated by EOI.
ATN UNT	Controller untalks and unlistens
ATN UNL	all devices.

The PUT Command

The PUT command is used to send data or device-specific commands to one or more instruments or peripheral devices. This command first asserts REN (remote enable) and unasserts NRFD (not ready for data). It then asserts ATN (attention) and sends UNT, UNL (untalk, unlisten) to assure that the bus is not active (no devices are talking or listening). The PUT command affects one or more instruments by sending a list of listen addresses with ATN, then a list of data items. EOI (end or identify) is asserted with the last data byte. This transaction is followed by ATN with the UNT, UNL messages and NRFD is released. Then ATN is unasserted.

The PUT command syntax used by the low-level interface driver is:

Syntax Form:

```
                 (expression        )[  (expression        )]
[line no.] PUT   {array expression  }[ ,{array expression  }]...
                 (string expression )[  (string expression )]
            INTO @expression,{expression [,expression]}
                             {array expression        }
                       [ ;{expression[,expression]}]
                       [  {array expression       }]...
```

Descriptive Form:

```
                 (source expression        )[  (source expression        )]
[line no.] PUT   {source array expression  }[ ,{source array expression  }]...
                 (source string expression )[  (source string expression )]
            INTO @IEEE 488 interface number,
                       {primary listen address [,secondary address]}
                       {listen and secondary address pairs         }
                       [ ;{primary listen address[,secondary address]}]
                       [  {listen and secondary address pairs        }]...
```

The source arguments for a PUT may be numeric expressions, array expressions, or string expressions. The number of bytes transferred and the order in which they are transferred is determined by the type of argument and the current data transfer mode. The specific effects are discussed below.

Type	**Mode**	**Description**
expression:	UNP:	The expression is evaluated and the integer part of the value, modulo 256, is transmitted.
	PAK LBF:	The expression is evaluated and the integer part of the value, modulo 256, is transmitted, followed by the integer part divided by 256, modulo 256.
	PAK HBF:	The expression is evaluated and the integer part of the value divided by 256, modulo 256, is transmitted, followed by the integer part of the value, modulo 256.
	HOG or UNH:	No effect.
array expression:	UNP:	Similar to an expression in UNP mode, except that one byte is transmitted for each value in the array expression.
	PAK LBF:	Similar to an expression in PAK LBF mode, except that a pair of bytes is transmitted for each value in the array expression.
	PAK HBF:	Similar to an expression in PAK HBF mode, except that a pair of bytes is transmitted for each value in the array expression.
	HOG or UNH:	No effect.
string expression:		One byte is transmitted on the data lines for each character in the string. PAK, UNP, HBF, LBF, HOG and UNH have no effect.

The expression following the at sign (@) is the interface number of the bus over which the data transfer takes place.

The list of expressions following the interface number indicates the addresses of the devices receiving the data. The elements of the list can be a numeric expression, a pair of numeric expressions, or an array expression. If a numeric expression or a pair of numeric expressions is used, the first expression is the primary listen address of a device. It must be between 32 and 62, inclusive. The optional second expression is the secondary address of the device. If used, it must be between 96 and 126, inclusive. Notice that the address pair is separated by a comma but that list items are separated by semicolons.

If an array expression is used, it must contain an even number of elements. Each pair must be a primary listen address and a secondary address. If a negative number is specified for a primary address in the address pair, the pair is skipped over when the addresses are sent. If a negative number is specified for the secondary address in the pair, only the primary address is sent. (These negative addresses are illegal except in an array.) Unless it is negative, the primary listen address must be between 32 and 62, while the secondary address must be between 96 and 126.

Bus Traffic: PUT example

Messages	**Comments**
ATN UNT	Controller untalks and
ATN UNL	unlistens all devices.
ATN talk address	Talk address from expression.
[ATN secondary address]	Secondary address from expression.
.	
.	
.	
<data byte>	Data bytes to instrument(s)
.	from source expressions.
.	
.	
<data byte> EOI	Terminated by EOI.
ATN UNT	Controller untalks and
ATN UNL	unlistens all devices.

An Example Using PUT and GET

Here is a routine that uses PUT and GET to read the settings of a TEKTRONIX 7912AD Programmable Digitizer mainframe and two plug-ins: a 7A16P Programmable Amplifier and a 7B90P Programmable Time Base. The settings are "remembered" by storing them in a file. They can then be changed as necessary. When you want to return the devices to the stored settings, executing a second routine reads the file and restores the settings with another PUT statement.

In this example, all three devices have a primary listen address of 33 and a primary talk address of 65. They are distinguished by their sequential secondary addresses of 96 for the mainframe, 97 for the amplifier, and 98 for the time-base.

```
100 LOAD "GPI.SPS"
110 FN$="SETFIL"
120 SIFTO @0,3000
130 STERMC @0,""
140 REM READ AND STORE THE SETTINGS
150 CANCEL FN$
160 OPEN #1 AS FN$ FOR WRITE
170 FOR I=0 TO 2
180 PUT "SET?" INTO @0,33,96+I
190 GET S$ FROM @0,65,96+I
200 WRITE #1,S$
210 NEXT I
220 CLOSE #1
230 REM SETTINGS CHANGED IN REST OF PROGRAM
 .
 .
 .
500 REM RESTORE SETTINGS
510 OPEN #1 AS FN$ FOR READ
520 FOR I=0 TO 2
530 READ #1,S$
540 PUT S$ INTO @0,33,96+I
550 NEXT I
560 CLOSE #1
570 RETURN
```

Line 100 LOADs the IEEE 488 Interface driver, while line 110 defines the name of the storage file. Then line 120 sets the time-out value to 3000 milliseconds (three seconds) to give the mainframe time to respond to changes in the settings and to pass on the handshake to the plug-ins when the second routine is executed. Line 130 makes sure that the termination character string is null so that the data transfer of the settings information from the devices will terminate only on an EOI.

The file is filled in lines 15Ø through 22Ø. Line 15Ø CANCELs any previous file with the same name before line 16Ø OPENs the storage file for WRITE. Then from a loop, the settings for the mainframe, the amplifier, and the time base are queried, read, and stored in a file. A PUT command (line 18Ø) queries each device by sending it the string "SET?". Then a GET statement (line 19Ø) reads the settings into a single string which is simply written to the file. After the three strings are stored, line 22Ø CLOSEs the file.

The string that is sent and stored is in a format acceptable to the device as a multiple set command. When queried by the "SET?", each device returns the Header and argument for all of its settings. The individual settings are separated by semicolons. Also, the programmable plug-ins return a carriage return and linefeed after all but the last argument, while the mainframe sends a semicolon after the last argument. Returning this string to the device restores the settings.

In the restore routine, line 51Ø OPENs the file for READ. Then the FOR loop (line 52Ø through 55Ø) READs a stored string, and uses a PUT to send it as a command string to the appropriate device until all three strings have been read and sent. This restores the settings to what they were when the first routine executed. Finally, line 56Ø CLOSEs the file.

NOTE

One of the commands sent to the mainframe by line 54Ø of the second routine will be a graticule command of either GRAT ON or GRAT OFF. For release number 1 of the 7912AD firmware you must perform a digitize operation to initialize the firmware before you execute the second routine.

Transferring Program Control on Interrupt

The WHEN Command

The WHEN command is used to alter the normal sequence of program execution on the occurrence of an interrupt or an anticipated event. This command transfers program control to a specified user-written interrupt subroutine when the event occurs and the system priority is less than the priority given that subroutine by the WHEN command. The IEEE 488 Interface Driver can use WHEN statements to respond to these kinds of events: an instrument SRQ (service request), an ERR (error) condition, or an EOI (end or identify) condition.

The WHEN command syntax used by the low-level interface driver is:

Syntax Form:

[line no.] **WHEN** @expression **HAS** string expression [**AT** expression]
[**AS TASK** expression] **GOSUB** line number

Descriptive Form:

[line no.] **WHEN** @IEEE 488 interface number **HAS** driver-dependent interrupt specification
[**AT** priority level] [**AS TASK** task number] **GOSUB** line number

The expression following the at sign (**@**) is the interface number of the IEEE 488 interface being accessed.

The string expression followng the keyword **HAS** is the interrupt specification. For the interface driver, it must evaluate to one of three strings: "SRQ", "EOI", or "ERR".

The optional keyword **AT** and the expression following it specify the execution priority of the interrupt routine that is scheduled when the given interrupt occurs. The priority can be any number between Ø and 126, inclusive. If no priority is specified, a default priority of 51 is used. (TEK SPS BASIC operates at a default system priority of 5Ø.)

The expression following the keywords **AS TASK** is the task number for the interrupt routine. It can be any value between Ø and 126, inclusive. If the optional AS TASK and task number are omitted, the number of the current task (the task number associated with the WHEN command as it executes) is used.

The keyword **GOSUB** precedes the starting line number of the user-written interrupt routine -- the subroutine to which control passes when the interrupt occurs.

An **SRQ interrupt** is enabled for an IEEE 488 interface upon execution of an SRQ WHEN statement. Once an SRQ interrupt has occurred for a given instrument, the SRQ interrupt capability for that interface is disabled until it is re-enabled by the execution of a POLL command or a GETSTA command which reads the status byte of an instrument that is asserting SRQ. (Both the POLL and GETSTA commands assume that if bit 7 of the status byte is set, the device is asserting SRQ. When either command detects that a device is asserting SRQ, the SRQ interrupt is re-enabled.)

To ensure that the SRQ interrupt is re-enabled, the SRQ interrupt subroutine should use POLL or GETSTA to read the status bytes of the devices on the bus until at least one device is found to be asserting SRQ. If any device on the bus does not use bit 7 of the status byte to report an SRQ, the subroutine will need to re-enable the SRQ interrupt explicitly. This can be done by executing another SRQ WHEN statement.

An **EOI interrupt** is enabled for an IEEE 488 interface upon execution of an EOI WHEN statement. The EOI condition is detected by the IEEE 488 Interface when the EOI and DAV lines on the IEEE 488 Interface bus are asserted. The EOI interrupt is re-enabled when the interrupt occurs.

The **ERR interrupt** is enabled during initialization of BASIC when the system software is loaded. An ERR WHEN statement allows a BASIC program to transfer control to a specified interrupt routine when an error occurs during a data transfer with an IEEE 488 interface.

The ERR WHEN statement should assign a different task number to the interrupt routine than the task number associated with the data transfer task. (By default, BASIC RUNs as task zero.) This should be done because the BASIC error handler (explained in Section 8) aborts all parts of a task in which a fatal error happens. If a fatal error occurs in a data transfer and the error interrupt routine has the same task number as the data transfer, the interrupt routine does not execute.

When a data transfer error occurs, an error message is issued and the error status condition is saved. This happens whether or not an ERR WHEN has been executed. The error status is accessible through the GIFES (Get InterFace Error Status) command. After an error interrupt occurs, the ERR interrupt is re-enabled.

The IGNORE Command

The IGNORE command cancels the actions of one or more WHEN commands. After a WHEN is canceled, the occurrence of the interrupt condition specified in the WHEN no longer causes the interrupt subroutine to be scheduled for execution. However, IGNORE has no effect on the execution of user-written interrupt routines already scheduled for execution.

The SRQ and the EOI interrupts are enabled by a WHEN statement so the actual interrupt capability can be disabled by an IGNORE statement. The ERR interrupt is enabled when BASIC is initialized and it cannot be disabled by an IGNORE statement.

The IGNORE command syntax used by this IEEE 488 Interface driver is:

Syntax Form:

```
                  (@expression [,(string expression)] )
[line no.] IGNORE <              [ (TASK expression  )] >
                  (TASK expression                      )
                  (All                                  )
```

Descriptive Form:

```
                  (@IEEE 488 interface number [,(driver-dependent interrupt specification)] )
[line no.] IGNORE <                           [ (TASK task number                        )] >
                  (TASK task number                                                         )
                  (All interrupt conditions for all interfaces                              )
```

The expression following the at sign (@) is the number of the IEEE 488 interface being accessed.

The expression following the keyword **TASK** is the task number associated with the user-written interrupt routine specified in the WHEN statement. The string expression specifies an interrupt condition which, for this driver, must evaluate to one of three strings: "SRQ", "EOI", or "ERR".

If only the interface number is specified, the actions of all WHEN statements specifying that interface number are canceled. If the interface number is followed by the optional interrupt condition string, only the WHEN that specifies that interface number and that interrupt condition is canceled. If the interface number is followed by the optional keyword TASK and the task number, all WHENs that specify both that interface number and that task number are canceled.

IF only the keyword TASK and the task number are specified, all WHENs specifying that task number are canceled.

If the keyword ALL is specified, the actions of all WHEN statements are canceled.

IEEE 488 Interface Driver Commands

In the rest of this section, each of the low-level IEEE 488 Interface driver commands is discussed in detail. To make them easier to find, the command descriptions are presented in alphabetical order. Each description includes statement examples, syntax information, and a general discussion of what the command does. A simple table showing typical IEEE 488 bus traffic is provided with those commands that generate a series of bus operations. Some of the discussions also contain example programs. The notation used in the syntax and descriptive forms is the same used for the System Commands described in Section 4.

Requirements:

All the IEEE 488 Interface driver commands require that the interface driver module "GPI.SPS" is LOADed and that the IEEE 488 capabilities are retained when the system is loaded and initialized.

Expression Evaluation:

These commands allow you to enter expressions for the interface number and the bus addresses. Any expression for an interface number is evaluated and **rounded** to an integer. That integer must be between 0 and 3, inclusive. Any expression used as a bus address is evaluated and the integer part modulo 256 is used as the address. (Another way to say this is that the number is **truncated** to a 16-bit integer of which the low-order eight bits are used for the address.) For bus addresses, the result of the expression must lie in one of the following ranges:

address	**range**
primary listen	32-62
primary talk	64-94
secondary	96-126

Command Summaries:

Below is a summary of the fifteen IEEE 488 Interface driver commands:

GETSTA	Gets the status byte of a bus-connected device.
GIFES	Gets the error status of the specified interface.
IFDTM	Sets the data transfer mode of the specified interface.
POLL	Performs a serial poll of the bus-connected devices.
PPOLL	Performs a parallel poll of the bus-connected devices.
RASCII	Reads ASCII data from a bus-connected device and stores it in the specified variable(s).
RBYTE	Reads a single byte of data through an IEEE 488 interface into a numeric variable.
READBINARY	Acquires an array of data sent from a Tektronix instrument in binary block format.
SIFCOM	Sends IEEE 488 addressed and universal commands to bus-connected devices.
SIFLIN	Controls the IEEE 488 interface lines.
SIFTO	Sets the interface time-out value.
STERMC	Designates the termination character string for ASCII data read into a string variable by a GET or RASCII statement.
TIFL	Reads the current setting of the control lines of an IEEE 488 bus.
WASCII	Sends ASCII data to a bus-connected device. Converts numeric data to an ASCII string before sending it.
WBYTE	Sends a byte of data through an IEEE 488 interface to the bus.

GETSTA (Nonresident)

Examples:

```
190 GETSTA @1,X,65
550 GETSTA @N,SB(N),TA(N),SA(N)
```

Syntax Form:

```
[line no.] GETSTA @expression,variable,expression[,expression]
```

Descriptive Form:

```
[line no.] GETSTA @IEEE 488 interface number, target variable for status byte,
                  talk address[,secondary address]
```

Purpose:

To get the status byte of a bus-connected instrument.

Discussion:

GETSTA (GET STAtus byte) returns the status byte of a bus-connected device. Status data may be obtained from a device even when that device is not asserting an SRQ (service request). If the device is asserting SRQ when its status is taken by a GETSTA command, the SRQ is cleared. Clearing SRQ allows subsequent SRQ interrupts to be recognized.

The default time-out value for a device not responding is 5 milliseconds. The time-out value can be changed by the SIFTO command.

The GETSTA command assumes that an instrument reports an SRQ by setting bit 7 of the status byte. When GETSTA detects that an instrument is asserting SRQ, it re-enables the SRQ interrupt. (The SRQ interrupt for an interface is enabled by executing an SRQ WHEN statement.)

Using the Command Syntax:

The expression following the at sign (@) is the number of the IEEE 488 interface to which the device is connected.

The variable specifies where the status byte of the designated device is to be stored.

The two expressions specify the address of the device. The first expression is the device primary talk address. The optional second expression may be used to send a secondary address if required. A primary talk address must be between 64 and 94, inclusive. A secondary address must be between 96 and 126, inclusive.

Bus Traffic:

ATN	UNT	Controller untalks and unlistens all
ATN	UNL	devices on the bus.
ATN	SPE	Then it enables a serial poll.
ATN	talk address	Talk address and optional
[ATN	secondary address]	secondary address sent to drive.
	<status byte>	Status byte sent from device.
ATN	UNT	Controller untalks device
ATN	SPD	and disables serial poll.

Application Example:

This routine uses GETSTA to return the status byte of a TEKTRONIX 7912AD Programmable Digitizer. It then shows how you might decode the status byte by making extensive use of VARTST to test which bits of the status byte are set.

The table below shows the status byte codes for the 7912AD. (To be consistent with the IEEE 488-1975 standard, the bits of the status byte are numbered from 1, not from Ø.) Bit 7 is tested for SRQ asserted, bit 6 for an error condition, and bit 5 for busy.

7912AD Status Byte Codes

Meaning:	**Bits:**							
	8	7	6	5	4	3	2	1
Remote Request	1	X	Ø	X	Ø	Ø	Ø	1
No Condition	Ø	Ø	Ø	X	Ø	Ø	Ø	Ø
Power Up	Ø	1	Ø	X	Ø	Ø	Ø	1
Operation Complete	Ø	X	Ø	X	Ø	Ø	1	Ø
Command Error	Ø	1	1	X	Ø	Ø	Ø	1
Execution Error	Ø	1	1	X	Ø	Ø	1	Ø
Internal Error	Ø	1	1	X	Ø	Ø	1	1
Power Fail Error	Ø	1	1	X	Ø	1	Ø	Ø

An X in the table means that the bit may be either a 1 or Ø. For this example, however, we assume that both Remote Request and Operation Complete have been programmed to assert SRQ. So, for this routine, a zero should appear in bit 7 of the status byte only for the No Condition code.

```
100 DIM ER$(4)
110 ER$(0)="ILLEGAL CODE FOR 7912AD"
120 ER$(1)="COMMAND ERROR"
130 ER$(2)="EXECUTION ERROR"
140 ER$(3)="INTERNAL ERROR"
150 ER$(4)="POWER FAIL ERROR"
160 DELETE 100,150
 .
 .
 .
500 GETSTA @N,SB,TA,SA
510 REM DECODE STATUS BYTE OF 7912AD
520 REM
530 REM TEST FOR BUSY
540 VARTST SB,"20",BZ\REM BINARY 10000
550 IF BZ=1 THEN PRINT "DEVICE BUSY"
560 REM TEST FOR SRQ
570 VARTST SB,"100",B\REM BINARY 1000000
580 IF B=0 THEN 960\REM NO SRQ
590 PRINT "SERVICE REQUEST"
600 REM TEST FOR NORMAL/ABNORMAL
610 VARTST SB,"40",B\REM BINARY 100000
```

```
620 IF B=0 THEN 720\REM NORMAL
630 REM ABNORMAL
640 REM DECODE, PRINT MESSAGE
650 REM FIND MODULO 8 VALUE OF STATUS BYTE
660 M8=SB-ITP(SB/8)*8
670 IF M8>4 THEN M8=0\REM LEGAL CODES:1-4
680 PRINT ER$(M8)
690 GOTO 960\REM RETURN
700 REM NORMAL
710 REM TEST FOR REMOTE REQUEST
720 VARTST SB,"200",B\REM BINARY 10000000
730 IF B=0 THEN 780\REM NOT REMOTE REQUEST
740 PRINT "REMOTE REQUEST"
750 GOSUB 2000\REM READ FRONT PANEL
760 GOTO 960\REM RETURN
770 REM TEST FOR POWER UP
780 VARTST SB,"1",B
790 IF B=0 THEN 830\REM NOT POWER UP
800 PRINT "POWER UP"
810 GOTO 960\REM RETURN
820 REM TEST FOR OPERATION COMPLETE
830 VARTST SB,"2",B
840 IF B=0 THEN 950\REM NOT OPERATION COMPLETE,SO ERROR
850 PRINT "OPERATION COMPLETE"
860 GOSUB 3000\REM ACQUIRE DATA
870 GOTO 960\REM RETURN
880 REM NO SRQ
890 REM TEST FOR NO CONDITION
900 VARTST SB,"357",B\REM BINARY 11101111
910 IF B=1 THEN 950\REM ERROR
920 PRINT "NO CONDITION"
930 GOTO 960\REM RETURN
940 REM ERROR
950 PRINT ER$(0)
960 RETURN
```

Early in the program, lines 100 to 150 fill an error message array. Then those lines are DELETEd to make room for data acquisition.

Line 500 returns the status byte (SB) of a 7912AD whose primary talk and secondary addresses are TA and SA. First, the routine checks the status byte for Busy by testing bit 5. The VARTST statement in line 540 sets a flag (BZ) to either 1 for Busy or 0 for not Busy. (This flag can be used by later routines, such as one to acquire data.) Next it checks for SRQ by testing bit 7. If bit 7 is set, the routine checks for normal or abnormal conditions by testing bit 6. If the condition is abnormal (bit 6 is set), the program looks at the last four bits by calculating the modulo 8 value of SB (line 660). It then prints one of the error messages before exiting.

When the condition is normal, the routine checks first for a Remote Request Code by testing bit 8 (line 720). (The 7912AD has only one remote request code, so there is no need to check bit 1 also.) If bit 8 is set, the routine transfers to a subroutine, such as one to read the front panel settings. (The example using PUT and GET earlier in this section shows how to read and store the front panel settings.) If bit 8 is not set, the routine tests for Power Up (line 780), and if not Power Up, it checks for Operation Complete (line 830). If the code is Operation Complete, the routine transfers to a subroutine to acquire data. If the code is not Operation Complete, since there are no more codes with SRQ asserted to test, an error message is issued before the routine exits.

When SRQ is not asserted, the routine goes to line 900 to test for No Condition. Here, since we expect all bits (except possibly bit 5) to be zero, we use VARTST to compare SB with binary 11101111. If any of the bits (except bit 5) are set, VARTST makes B equal 1 for true. Since B must be 0 for a No Condition code, if B is not 0, the routine prints an error message before exiting.

GIFES (Nonresident)

Examples:

```
710 GIFES @N+1,A(J)
120 GIFES @2,ES
```

Syntax Form:

[line no.] **GIFES** @expression,variable

Descriptive Form:

[line no.] **GIFES** @IEEE 488 interface number,target variable

Purpose:

To get the error status of the specified interface.

Discussion:

GIFES (Get InterFace Error Status) acquires the error status for the specified IEEE 488 interface and returns it in the target variable. This error is for the interface itself and not for the bus-connected devices. [GIFES reads bits 12, 13, and 14 of the Interface Status Register (ISR).]

After GIFES executes, the variable holds one of the following error status codes:

0	**NO ERROR**
1	**DMA** error (The Bus Address Register (BAR) used by the interface for DMA operations has overflowed, or the interface has attempted to reference nonexistent memory.)
2	**WRITE** error (No listener was enabled on the bus.)
3	**DMA** and **WRITE** errors
4	**WRITE TIMING** error (A hardware timing error has occurred.)
5	DMA and **WRITE TIMING** errors

6 **WRITE** and **WRITE TIMING** errors
7 **DMA**, **WRITE**, and **WRITE TIMING** errors

The three types of errors indicated by the code are the errors that can cause execution of a user-written error-handling subroutine if an ERR WHEN has been set up. If an ERR WHEN has not been executed, these errors are handled as fatal errors.

When an error occurs, the error status is stored in the Interface driver. The errors are not stacked since there is room for only one error status value for each interface. The latest error overwrites any previous error code. Executing the GIFES command clears the error status of the specified interface. A subsequent GIFES statement specifying the same interface number will return a zero or any new error that may have occurred.

Using the Command Syntax:

The expression following the at sign (**@**) is the number of the IEEE 488 interface being accessed.

The variable specifies where the error status of the interface is to be stored.

Bus Traffic: None.

Application Example:

This example prints one or more error messages when an interface error occurs for interface number N.

```
100 WHEN @N HAS "ERR" AT 126 AS TASK 5 GOSUB 900
 .
 .
 .
900 GIFES @N,E
910 GOTO E OF 920,930,920,930,920,930,920
920 PRINT "DMA ERROR"
930 GOTO E OF 950,940,940,950,950,940,940
940 PRINT "WRITE ERROR"
```

```
950 GOTO E OF 970,970,970,960,960,960,960
960 PRINT "WRITE TIMING ERROR"
970 RETURN
```

First, line 100 sets up a WHEN condition that allows an interrupt on an interface error. (Remember that the error routine should have a different task number than the data transfer routine.) If an error occurs, the program jumps to the high priority task at line 900. There a GIFES statement returns the error code in E. Then three computed GOTO statements decode E so that the appropriate error messages are printed. Since line 900 should only be executed if there is an error, E should never equal zero.

IFDTM (Nonresident)

Examples:

```
170 IFDTM @1,"UNP"
250 IFDTM @N,"PAK","HOG","LBF"
```

Syntax Form:

[line no.] **IFDTM** @expression,string expression[,string expression] ...

Descriptive Form:

[line no.] **IFDTM** @IEEE 488 interface number, specification of mode of data transfer [,specification of mode of data transfer] ...

Purpose:

To set the data transfer mode of the specified interface.

Discussion:

IFDTM (InterFace Data Transfer Mode) specifies the mode of data transfer to be used by the IEEE 488 interface during the execution of a PUT, GET, or READBINARY command. This applies to numeric data only.

The legal strings are "PAK" or "UNP", "HOG" or "UNH", and "LBF" or "HBF" where:

String:	**Meaning:**	**Comments:**
PAK	Packed: Data is transferred as 16-bit, 2's complement numbers. Two 8-bit bytes are transferred for each value.	"HBF" or "LBF" determines which byte is the most significant.
UNP	Unpacked: Data is transferred as an unsigned 8-bit byte.	"HBF" or "LBF" has no effect.
HOG	Monopolize: Processor is held up while DMA "hogs" the bus to complete data transfer (fastest DMA).	For the CP4100/IEEE 488 interface, hog mode is really "burst" mode to allow the processor to refresh volatile memory.
UNH	Unhog: Data transfer is completed during the inactive periods between processor cycles (cycle-stealing)	
LBF	Low byte first: the least significant 8 bits of a 16-bit value are transferred first.	**Used only in Packed mode data transfers.** These specify which of the two bytes transferred is the most significant.
HBF	High byte first: the most significant 8 bits of a 16-bit value are transferred first.	

Upon loading the driver, the default values are: UNP, UNH, HBF.

Using the Command Syntax:

The expression following the at sign (**@**) is the number of the IEEE 488 interface whose data transfer mode will be set.

The list of string expressions specify the data transfer mode. Since the legal strings are mutually exclusive pairs, a IFDTM statement should include, at most, only one of each pair.

Bus Traffic: None

POLL (Nonresident)

Examples:

```
770 POLL @2,SB,PA,SA;64,96;64,97;64,98
530 POLL @N,X,Y,Z
480 POLL @3,SB,PA,SA;64;65;66
610 POLL @2
```

Syntax Form:

```
[line no.] POLL @expression,variable,variable,variable [;{array expression             }] ...
                                                         {expression[,expression]     }
```

Descriptive Form:

```
[line no.] POLL @IEEE 488 interface number, target variable for status byte,
                target variable for primary address, target variable for secondary address

                [;{talk and secondary address pairs }] ...
                  {talk address[,secondary address] }
```

Purpose:

To perform a serial poll of the bus-connected devices through the specified interface.

Discussion:

POLL solicits information from bus-connected devices by performing a serial poll on the bus. Polling stops with the first device to affirm a service request (SRQ).

The POLL command assumes that an instrument reports an SRQ by setting bit 7 or the status byte. When POLL detects that an instrument is asserting SRQ, it re-enables the SRQ interrupt. (An SRQ interrupt for an interface is enabled by executing an SRQ WHEN statement.)

If a list of primary or primary and secondary addresses is specified in the command statement, only those addresses are sequentially polled. If a list is not specified, the command sequentially polls all addresses on the bus until a device responds with an affirmative request for service. This is the simplest way to specify the command, but it is very inefficient. If no device responds, the command could serially poll all 961 possible addresses (31 possible primary talk addresses times 31 possible secondary addresses). The default time-out value for addresses that do not respond is 5 milliseconds. (The time-out value can be changed with the SIFTO command.)

The status byte and the primary and secondary talk addresses of the first device to respond to the poll with an affirmative request for service (SRQ) are returned in the designated variables. When no instruments or devices respond with an affirmative request for service, the values returned for the status byte and the primary and secondary addresses are zero.

Using the Command Syntax:

The expression following the at sign (**@**) is the number of the IEEE 488 interface to be polled.

The status byte of the first device to respond to the poll with an affirmative request for service (SRQ) is returned in the first variable.

The primary talk address of the responding bus-connected device is returned in the second variable. The secondary address of the responding bus-connected device is returned in the third variable.

The optional list of expressions represents a sequence of primary talk addresses (and secondary addresses if required) for the devices to be polled. The list elements, which may be either array expressions or numeric expressions, are separated by semicolons (;). When a list is supplied, only those addresses are POLLed. If the list is omitted, all possible addresses on the bus may be POLLed.

If an array expression is specified, the resultant array must contain an even number of elements. These elements must be pairs of addresses; a primary talk address followed by a secondary address. If a negative number is specified for a primary address in an array, the address pair is skipped over and the next address pair is used in the poll. If a negative number is specified for a secondary address, only the primary address is used.

If numeric expressions are used, a comma separates the primary talk address from the optional secondary address. When numeric expressions are used, negative numbers are illegal.

Expressions for primary talk addresses must evaluate to numbers between 64 and 94, while secondary addresses must be between 96 and 126. An exception to this is when the address is specified in an array expression and a negative value is given to cause POLL to ignore (skip over) either the address pair or the secondary address, as explained above.

Bus Traffic:

```
         ATN  UNT                    Controller untalks and unlistens
         ATN  UNL                    all devices on the bus. Then it
         ATN  SPE                    enables a serial poll.
         ATN  talk address           First address (pair) Time-out wait
        [ATN  secondary address]     if device does not respond.
                .
                .
                .
         ATN  talk address           Next address (pair).
        [ATN  secondary address]     If no response,
                .                    time-out wait and continue
                .                    until a device responds with
                .                    a status byte indicating SRQ.
         ATN  talk address           Address (pair) of first
        [ATN  secondary address]     instrument that responds. Status
              <status byte>          byte from responding instrument.
         ATN  UNT                    Controller untalks and then
         ATN  SPD                    disables serial poll.
```

Talk addresses (and secondary addresses) are sent from the list specified in the command, or in sequential order if no list is given. Addresses continue to be sent until a device responds with a status byte that indicates it asserted SRQ or the list is exhausted.

Application Example:

The best way to explain how to use the POLL command is to give a specific example. Our example system contains four TEKTRONIX 7912AD Programmable Digitizers, a programmable signal generator, and a programmable power supply. Each 7912AD has two plug-ins: a TEKTRONIX 7A16P Programmable Amplifier and a TEKTRONIX 7B90P Programmable Time Base. We have configured the system so that the four 7912ADs have primary talk addresses of 64,65,66, and 67. Each 7912AD mainframe has its plug-ins set to the mainframe primary talk address. To distinguish the mainframe and plug-ins, the required secondary addresses are set to 96 for the mainframes, 97 for the amplifiers, and 98 for the time bases. The signal generator and the power supply have primary talk addresses 68 and 69, respectively, and need no secondary addresses. Given such a system, this routine shows how you might conduct a serial poll, and use the information returned by POLL:

```
200 Q=3
210 DIM AD$(2),MF(Q,1),AM(Q,1),TB(Q,1)
220 REM FILL MESSAGE ARRAY
230 AD$(Ø)="MAINFRAME"
240 AD$(1)="AMPLIFIER"
250 AD$(2)="TIME BASE"
260 REM FILL ADDRESS ARRAYS
270 FOR I=Ø TO Q
280 MF(I,Ø)=I+64
290 AM(I,Ø)=I+64
300 TV(I,Ø)=I+64
310 NEXT I
320 MF(Ø:Q,1)=96
330 AM(Ø:Q,1)=97
340 TB(Ø:Q,1)=98
350 DELETE 200,340
360 WHEN @N HAS "SRQ" AT 126 GOSUB 1010
 .
 .
 .
1000 REM POLL ROUTINE
1010 POLL @N,SB,PA,SC;MF;AM;TB;68;69
1020 REM CHECK FOR SRQ
1030 IF SB<>Ø THEN 1060\REM SRQ TRUE
1040 PRINT "SRQ DETECTED BUT NO RESPONSE TO POLL"
1050 GOTO 1100\REM RETURN
```

```
1Ø6Ø PRINT "SRQ ON INTERFACE ";N;" FROM";
1Ø7Ø REM DETERMINE DEVICE TYPE
1Ø8Ø REM AND DECODE STATUS BYTE
1Ø9Ø GOSUB PA-63 OF 2ØØØ,2ØØØ,2ØØØ,2ØØØ,3ØØØ,4ØØØ
11ØØ RETURN
  .
  .
  .
2ØØØ PRINT "7912AD #";PA-63,AD$(SC-96)
2Ø1Ø REM DECODE STATUS BYTE FOR
2Ø2Ø REM 7912AD MAINFRAME, AMPLIFIER, OR TIME BASE
2Ø3Ø GOSUB SC-95 OF 21ØØ,24ØØ,27ØØ
2Ø4Ø RETURN
  .
  .
  .
3ØØØ PRINT "SIGNAL GENERATOR"
3Ø1Ø REM DECODE STATUS BYTE
  .
  .
  .
35ØØ RETURN
  .
  .
  .
4ØØØ PRINT "POWER SUPPLY"
4Ø1Ø REM DECODE STATUS BYTE
  .
  .
  .
45ØØ RETURN
```

Early in the program, lines 23Ø to 25Ø fill a message array (AD$). Then, to allow us to use arrays in the list of addresses in the POLL statement, lines 27Ø through 34Ø fill three arrays: one for the mainframe address pairs (FM), one for amplifier address pairs (AM), and one for time base address pairs (TB). You can picture the three arrays like this:

Mainframe address array		Amplitude address array		Time Base address array	
primary	secondary	primary	secondary	primary	secondary
64	96	64	97	64	98
65	96	65	97	65	98
66	96	66	97	66	98
67	96	67	97	67	98

Three separate arrays are set up for clarity. Also by changing the value of Q, you could fill the arrays for a system with a different number of 7912ADs.

After these lines execute, they are deleted (but not the array they fill) since the code is no longer needed. Then the "SRQ" WHEN is set up on line 360 with a priority of 126. After line 360 executes, when a bus-connected device asserts SRQ, the program control transfers to the serial poll subroutine.

The routine starts in line 1010 with a POLL statement. The POLL specifies that only the addresses in the three arrays plus the addresses of the two additional devices are to be polled. Line 1030 checks for a possible error (such as noise on the bus). Then, because we configured our system so simply, we can use a computed GOSUB (line 1090) to determine which routine to use to process the status byte (SB) returned by POLL. If POLL returns a primary address (PA) of 64 through 67, the program goes to the 7912AD routine (line 2000). A primary address of 68 sends it to line 3000 which decodes the status byte of the signal generator, while a 69 passes control to line 4000 which decodes the status byte of the power supply.

In the 7912AD routine, we use the secondary address returned by POLL (SC) in a second computed GOSUB. Line 2030 transfers control to the appropriate subroutine to decode the status byte of a mainframe, amplitude, or time base. (An example of how you might decode the status byte of the mainframe is shown with the GETSTA command.)

PPOLL (Nonresident)

Examples:

```
910 PPOLL @J,X(J)
790 PPOLL @0,A
```

Syntax Form:

[line no.] **PPOLL** @expression,variable

Descriptive Form:

[line no.] **PPOLL** @IEEE 488 interface number, target variable

Purpose:

To perform a parallel poll on the specified interface bus.

Discussion:

PPOLL (Parallel POLL) performs a parallel poll on the bus and returns in the designated variable a status-descriptive information code that is read from the data lines.

NOTE

The devices on the IEEE 488 bus must have been previously configured to respond to the PPOLL command. This can be done with the SIFCOM command.

Using the Command Syntax:

The expression following the at sign (**@**) is the number of the IEEE 488 interface through which to perform the parallel poll. The variable is used to store the information returned from the parallel poll.

Bus Traffic:

ATN EOI	ATN and EOI sent by controller to solicit parallel poll.
<data byte>	The data byte sent is a composite of one bit of status from each device with an SRQ that has been set-up to respond to a parallel poll.

RASCII (Nonresident)

Examples:

```
150 RASCII A$ FROM @N,TA,SA
880 RASCII X,Y FROM @0,65
570 RASCII X,A$ FROM @1
```

Syntax Form:

```
                    (variable        ) [ (variable        ) ]
[line no.] RASCII   {array           } [,{array           } ] ...
                    (string variable ) [ (string variable ) ]

                    FROM @expression[,expression[,expression]]
```

Descriptive Form:

```
                    (target variable        ) [ (target variable        ) ]
[line no.] RASCII   {target array           } [,{target array           } ] ...
                    (target string variable ) [ (target string variable ) ]

                    FROM @IEEE 488 interface number[,talk address[,secondary address]]
```

Purpose:

To read ASCII data from a bus-connected device and store it in the specified variables.

Discussion:

RASCII (Read ASCII) reads ASCII data from a bus-connected device into numeric variables, arrays, or string variables.

When a numeric variable is specified, RASCII ignores characters until it receives a legal numeric string character. (Legal characters include the plus sign (+), the minus sign (-), a decimal point (.), the letter E, and the numbers Ø through 9.) RASCII retains this numeric string character and continues reading characters into the numeric string until a nonnumeric character or an EOI is received. This last character is not retained. The numeric string is then converted to either a floating-point or an integer number depending on the type of the variable in which it is stored. RASCII uses the same routine as the INPUT command to convert a numeric string to a number.

When a string variable is specified, RASCII reads characters into the string variable until an EOI or the termination character string is received. The termination character(s) are not stored in the string. (To set the termination string use the STERMC command.)

The bus traffic depends on whether the RASCII statement specifies a talker or not. If the talk address is specified, RASCII untalks and unlistens all devices on the bus and enables the interface to listen without ATN asserted. It then sends the primary talk address and the optional secondary address with ATN asserted, making the specified device a talker. Then, after receiving the last byte of data, RASCII untalks and unlistens all devices on the bus and disables the interface to listen without ATN asserted. Finally, NRFD is unasserted.

If no talk address is specified, no interface messages are sent before or after the data transfer. The IEEE 488 bus is left in the same state that it was in before the execution of RASCII with the exception that NRFD is left asserted. This means that it is the user's responsibility to enable the interface to listen, to set up a talker before RASCII executes, and to leave the bus in an acceptable state. (See RBYTE for an example of how this can be done using SIFLIN and SIFCOM statements.)

The default time-out value for a device that does not respond is 5 milliseconds. The time-out value can be changed by the SIFTO command.

Using the Command Syntax:

The list of arguments preceding the keyword **FROM** specifies how the data is read and stored. If a numeric variable is specified, a numeric string is read according to the algorithm discussed above. The reading of the numeric string terminates with an EOI or a nonnumeric character. This string is then converted to a number and stored in the specified variable. If an array is specified, a numeric string is read, converted and stored for each array element according to the algorithm for a variable. If a string variable is specified, ASCII characters are read and stored in the string variable until terminated by an EOI or the termination string.

The expression following the at sign (@) is the number of the IEEE 488 interface through which the ASCII data is read.

The optional expressions following the interface number are the talk addresses of the device sending the data. The first expression is the primary talk address, which must evaluate to a number between 64 and 94. The optional second expression is the secondary address, which must evaluate to a number between 96 and 126.

Bus Traffic:

[ATN UNT ATN UNL ATN talk address [ATN secondary address]]	If optional talk address is specified, the controller untalks and unlistens a devices on the bus and sends the talk address and the optional secondary address.
<first ASCII data byte> . . . <last ASCII data byte>	Data bytes sent from talker.
EOI or other terminator	Data terminated by EOI or other termin
[ATN UNT ATN UNL]	If optional talk address is specified, the controller untalks and unlistens all devices on the bus.

RBYTE

Examples:

```
440 RBYTE @1,X
710 RBYTE @N,A(3)
```

Syntax Form:

```
[line no.] RBYTE @expression,variable
```

Descriptive Form:

```
[line no.] RBYTE @IEEE 488 interface number, target variable
```

Purpose:

To read a single byte of data through an IEEE 488 Interface into a variable.

Discussion:

RBYTE (Read BYTE) reads a single byte of data from the IEEE 488 interface and stores it in a numeric variable. If the talker does not send a byte of data within the time allowed by the time-out value, an error message is generated. (The driver has a default time-out of 5 milliseconds. This time-out value can be changed with the SIFTO command.)

After RBYTE executes, NRFD is left asserted.

Note that before the RBYTE command can actually acquire data, a bus-connected instrument must be waiting to talk. In addition, the specified IEEE 488 interface in the controller must be told to listen with the SIFLIN command.

Using the Command Syntax:

The expression following the at sign (@) is the number of the IEEE 488 interface through which the data is read. The byte of data is stored in the specified variable.

Bus Traffic:

<data byte>	Data byte from device previously addressed to talk.

Application Example:

This routine acquires ten bytes of data from a bus-connected instrument through an IEEE 488 interface into an integer array. It uses RBYTE in a FOR/NEXT loop to read and store the data one byte at a time.

```
100 INTEGER B(9)
110 SIFCOM @N,"UNT","UNL"
120 SIFLIN @N,"LENB"
130 SIFCOM @N,TA,SA
140 FOR I=0 TO 9
150 RBYTE @N,B(I)
160 NEXT I
170 SIFLIN @N,"ATNT","NRFDF","LDIS"
180 SIFCOM @N,"UNT"
```

Line 110 untalks and unlistens all the devices on the bus. Then line 120 enables interface number N to listen without ATN asserted. Next the SIFCOM in line 130 makes the instrument a talker by sending it its primary talk address (TA) and its secondary address (SA) with ATN asserted. When line 130 is finished and ATN is unasserted, the instrument, as talker, starts transferring data, a byte at a time. Lines 140 through 160 read the ten bytes of data into the array elements. Then line 170 asserts ATN, unasserts NRFD (since RBYTE leaves NRFD asserted), and disables the interface for listen without ATN asserted. ATN is asserted first in line 170 to prevent the talker from asserting DAV after NRFD is unasserted. Finally, line 180 untalks the instrument.

READBINARY (Nonresident)

Examples:

```
190 READBINARY X,Y FROM @0,66,96
740 READBI M FROM @I
565 READBINARY A FROM @N,TA(J)
```

Syntax Form:

```
[line no.] READBI {simple numeric variable} [,{simple numeric variable}] ...
                  {array                  }  [ {array                  }]
                  FROM @expression[,expression[,expression]]
```

Descriptive Form:

```
[line no.] READBINARY  target for binary block [,target for binary block]...
                       FROM @ IEEE 488 interface number
                            [,talk address [,secondary address]]
```

Purpose:

To acquire an array of data sent in binary block format from a Tektronix instrument.

Discussion:

READBINARY makes it very easy to read an array from a Tektronix instrument which transmits array data as a binary block, such as a 7912AD Programmable Digitizer. A binary block, as defined the Tektronix Codes and Formats Standard, consists of:

1. A percent sign (%).

2. A byte count. This is a two-byte binary integer which specifies the number of data bytes being transmitted including the checksum byte. The byte count is sent most significant byte first.

3. Zero or more data bytes.

4. A checksum byte.

READBINARY reads the entire binary block and any following delimiter, such as a comma or a semicolon, and stores just the binary block data in the specified predimensioned array or auto-dimensioned array. If the target array is contiguous, the data is stored using DMA (direct memory access). (In TEK SPS BASIC, all one-dimensional arrays, including autodimensioned arrays, are contiguous. A non-contiguous array can be formed only by specifying an array zone of a two-dimensional array in which the first subscript is zoned, e.g., A(Ø:511,2).)

The mode of the data transfer should be set by executing an IFDTM statement prior to executing READBINARY. The data transfer mode determines 1) if the data will be read as two bytes per element (packed) or as one byte per element (unpacked), 2) if packed data is interpreted as high byte first or low byte first, and 3) if a DMA transfer monopolizes the processor bus or not (hog or unhog).

The interaction between the data transfer mode set by the IFDTM command and the type of variables specified in the READBINARY command is shown below.

Type	**Mode**	**Action**
integer array:	UNP:	One byte is read from the bus and stored for each array element.
	PAK HBF:	Two bytes are read from the bus and stored for each array element. The value stored is equal to the first byte times 256, plus the second byte.
	PAK LBF:	Two bytes are read from the bus and stored for each array element. The value stored is equal to the second byte times 256, plus the first byte.
	UNH:	If the array is contiguous, transfer is via normal Direct Memory Access (DMA) protoco

	HOG:	If the array is contiguous, a DMA transfer is accomplished with the interface having exclusive access to the controller bus. No intervention is allowed. (For the CP4100/IEEE 488 interface, hog mode is really "burst" mode to allow the processor to refresh the volatile memory.)
simple numeric variable:	UNP	Same as for an integer array except the variable is first autodimensioned to an integer array whose size equals the number of data bytes in the binary block.
	PAK HBF: or PAK LBF:	Same as for an integer array except the variable is first autodimensioned to an array whose size equals half the number of data bytes in the binary block.
	UNH:	Same as for an integer array.
	or	
	HOG:	Same as for an integer array.
floating-point array:	UNP:	One byte is read from the bus and its floating-point representation is stored for each element of the array.
	PAK HBF:	Two bytes are read from the bus and stored for each array element. The value stored is the floating-point representation of the first byte times 256, plus the second byte.
	PAK LBF:	Two bytes are read from the bus and stored for each array element. The value stored is the floating-point representation of the second byte times 256, plus the first byte.
	UNH:	Same as for an integer array.
	HOG:	Same as for an integer array.

The bus traffic depends on whether the READBINARY statement specifies a talker or not. If the talk address is specified, READBINARY untalks and unlistens all devices on the bus and enables the interface to listen without ATN asserted. It then sends the primary talk address and the optional secondary address with ATN asserted, making the specified device a talker. Then, after receiving the last byte of data, READBINARY untalks and unlistens all devices on the bus and disables the interface to listen without ATN asserted. Finally, NRFD is unasserted.

If no talk address is specified, no interface messages are sent before or after the data transfer. The IEEE 488 bus is left in the same state that it was in before the execution of READBINARY with the exception that NRFD is left asserted. This means that it is the user's responsibility to enable the interface to listen, to set up a talker before READBINARY executes, and to leave the bus in an acceptable state. (See RBYTE for an example of how this can be done using SIFLIN and SIFCOM statements.)

The default time-out value for a device that does not respond is 5 milliseconds. The time-out value can be changed by the SIFTO command.

Using the Command Syntax:

The list of arguments preceding the keyword **FROM** specifies the number of binary blocks to be read and where the data is to be stored. One binary block is read for each argument in the list. Each argument receives all the array data in one binary block transmission, i.e., all the data between the byte count and the checksum.

If an argument is a simple numeric variable, it is autodimensioned to an integer array whose size is determined by the byte count and by the data transfer mode (packed or unpacked). If the mode is packed, two bytes are read for each array element, so the array size is half the byte count, minus 1. (The byte count includes the checksum byte.) If the mode is unpacked, one byte is read for each array element, so the array size is 1 less than the byte count.

If the argument is an array, the data is read directly into the array until either the array is full or the last data value in the binary block is sent. When the array size does not match the amount of the data sent, a warning I19 error is issued. If the array is filled before the last value is sent, the remaining data in the binary block is read and discarded.

For example, the sequence of statements:

```
DELETE X,Y
READBINARY X,Y FROM @N,TA,SA
```

reads one binary block into autodimensioned array X and another binary block into autodimensioned array Y. However, if an attempt is made to read the two binary blocks into three dimensioned arrays, A, B, and C, by a statement like:

```
READBINARY A,B,C FROM @N,TA,SA
```

An I19 error would be issued and array C would not receive any new data. Also, if the size of A is too small to hold all the data from the first binary block, the data that does not fit into A is read and discarded. It is not read into B. The data from the second binary block is read into B. If the size of B does not match the amount of data sent, another I19 error is issued.

When the binary block is empty, i.e., no data is sent between the byte count and the checksum, a warning I21 error is issued. This can occur, for instance, when the main intensity control of the 7912AD is set too low to allow any values to be stored into the verticals array. If the target is a simple numeric variable, it is **not** autodimensioned to an array; if the target is an array, its data is not changed.

The expression following the at sign **(@)** is the number of the IEEE 488 interface through which the data is read.

The optional expressions following the interface number are the talk addresses of the device sending the data. The first expression is the primary talk address, which must evaluate to a number between 64 and 94. The optional second expression is the secondary address, which must evaluate to a number between 96 and 126.

Bus Traffic:

[ATN UNT ATN UNL ATN talk address [ATN secondary address]]	If optional talk address is specified, the controller untalks and unlistens all devices on the bus and sends the talk address and the optional secondary address.
<first data byte> . .	Data bytes sent from talker, starting with a % and the byte count.
EOI<last data byte>	Data terminated by EOI or other terminator.

$\begin{bmatrix}\text{ATN UNT}\\\text{ATN UNL}\end{bmatrix}$	If optional talk address is specified, the controller untalks and unlistens all devices on the bus.

Application Example:

By using the READBINARY command, you can readily acquire the pointers array and the verticals array from a TEKTRONIX 7912AD Programmable Digitizer. For example, if the primary listen, primary talk, and secondary address of a 7912AD are LA, TA, and SA and the interface number is N, this routine reads that data into two autodimensioned arrays PT and VT.

```
110 PUT "DIG DAT;READ PTR,VER" INTO @N,LA,SA
120 IFDTM @N,"PAK","HBF"
130 DELETE PT,VT
140 READBINARY PT,VT FROM @N,TA,SA
```

Line 110 tells the 7912AD to digitize the input signal and to prepare to send the pointers and verticals arrays. Then line 120 sets the data transfer mode to packed and high byte first. Next, line 130 DELETEs the targets PT and VT to make sure they are not already arrays. Finally, line 140 stores the pointers array in autodimensioned integer array PT and the verticals array in autodimensioned integer array VT.

SIFCOM (Nonresident)

Examples:

```
150 SIFCOM @1,"DCL"
740 SIFCOM @N,X
230 SIFCOM @0,PA,SA,C$
```

Syntax Form:

```
                                 ( expression        ) [  ( expression        ) ]
[line no.] SIFCOM @expression, < array expression  > [ ,< array expression  > ]...
                                 ( string expression ) [  ( string expression ) ]
```

Descriptive Form:

```
                                                  ( source expression              )
[line no.] SIFCOM @IEEE 488 interface number,    < source array expression        >
                                                  ( interface command specification )

                     [  ( source expression               ) ]
                     [ ,< source array expression         > ] ...
                     [  ( interface command specification ) ]
```

Purpose:

To send IEEE 488 addressed and universal commands to bus-connected devices.

Discussion:

SIFCOM (Send InterFace COMmand) can send universal commands, addressed commands, and device addresses over the bus. The command or address is sent over the data lines while ATN is asserted. ATN is unasserted at the conclusion of SIFCOM.

The default time-out value for a device that does not respond is 5 milliseconds. This time-out value can be changed by the SIFTO command.

The IEEE 488 commands that can be sent by SIFCOM are:

Command	**Meaning**	**Comment**
UNL	UNListen	Disables listen mode of all previous listeners.
UNT	UNTalk	Disables talk mode of previous talker.

Universal Commands

LLO	Local LockOut	
DCL	Device CLear	Universal commands affect
PPU	Parallel Poll Unconfigure	all instruments on the
SPE	Serial Poll Enable	bus.
SPD	Serial Poll Disable	

Addressed Commands

GTL	Go To Local	Sent to device(s) addressed
SDC	Selective Device Clear	to listen.
GET	Group Execute Trigger	
PPC	Parallel Poll Configure	(PPC requires secondary command to Parallel Poll Enable.)
TCT	Take ConTrol	Sent to a device addressed to talk to make that device the controller-in-charge. (However, most of the driver commands requir that the Interface driver remain tl Controller-in-charge.)

UNL, UNT, and the universal commands do not require that an address be sent with the command. For example, to untalk and unlisten all devices on interface number N and set them to local lockout use:

```
SIFCOM @N,"UNT","UNL","LLO"
```

The addressed commands require that the device(s) be specified by address. For example, where LA is the primary listen address and SA is the secondary address of a device, the statement:

```
SIFCOM @N,LA,SA,"GTL"
```

sends GTL to the addressed device.

Using the Command Syntax:

The expression following the at sign (@) is the number of the interface being accessed.

The commands and addresses to be sent are specified by the list of expressions following the interface number. If a string expression is used, it must evaluate to one of the twelve IEEE 488 commands discussed earlier. If a numeric expression is used, it is evaluated and its integer part (modulo 256) is sent. If an array expression is used, the integer part (modulo 256) of each element is sent. Since ATN is asserted by SIFCOM, sending a talk address sets up an external talker, while sending a listen address sets up an external listener.

Bus Traffic

```
ATN     <data byte>    Controller sends commands or
 .           .         addresses as specified.
 .           .
 .           .
ATN     <data byte>
```

Application Example:

SIFCOM can be used to set up an external data transfer between two devices. In this example TA is the primary talk address of one device while LA is the primary listen address of another device. N is the interface number. Assuming that the talker asserts EOI with the last byte of data sent, this routine can be used.

```
100 SIFCOM @N,"UNT","UNL"
110 FL=0
120 WHEN @N HAS "EOI" GOSUB 200
130 SIFCOM @N,TA,LA
140 IF FL=0 THEN 140
150 PRINT "TRANSFER COMPLETE"
160 RETURN
200 IGNORE @N,"EOI"
210 FL=1
220 RETURN
```

First, line 100 untalks and unlistens all devices on the bus. Then a flag (FL) is set to 0 and line 120 sets up a transfer to a subroutine when an EOI is sent. Next, the SIFCOM in line 130 sends the talk and listen addresses to make one device a talker and one a listener. While the data transfer takes place, the program waits in an infinite loop at line 140. When EOI is sent, the program jumps to the subroutine at line 200 which cancels the WHEN and sets FL to 1. On returning to line 140, since FL is no longer 0, the message is printed and the routine terminates.

SIFLIN (Nonresident)

Examples:

```
550 SIFLIN @4,"IFC"
140 SIFLIN @N,A$,B$
```

Syntax Form:

```
[line no.] SIFLIN @expression,string expression[,string expression] ...
```

Descriptive Form:

```
[line no.] SIFLIN @IEEE 488 interface number,
                  specification of how interface line is to be set
                  [,specification of how interface line is to be set] ...
```

Discussion:

To control the IEEE 488 interface lines.

Discussion:

SIFLIN (Set InterFace LINes) lets the specified interface assert or unassert a control line. It can also enable or disable the interface to listen without ATN. [SIFLIN sets or clears the appropriate bit of the Bus Control Register (BCR) in the IEEE 488 interface.]

SIFLIN allows you to control the interface at a very low level. Most of the interface driver commands take care of the control lines for you. An exception is the RBYTE command which does not listen enable the interface or listen disable the interface when done. You must use SIFLIN with "LENB" or "LDIS" to do either of these when using RBYTE. (See the RBYTE discussion for an example.) The rest of the legal strings are included for those who want to write their own driver in BASIC.

Using the Command Syntax:

The expression following the at sign (@) is the number of the IEEE 488 interface that is to be set.

The string expression specifies the action to be taken. It must evaluate to one of the legal string mnemonics listed below. The string that sets an IEEE 488 management or control line (except IFC) is the mnemonic for the line plus a "T" for True (asserted) or an "F" for False (unasserted).

Mnemonic		Meaning
IFC		Interface Clear
RENT	RENF	Remote Enable (True or False)
EOIT	EOIF	End Or Identify (True or False)
ATNT	ATNF	Attention (True or False)
SRQT	SRQF	Service Request (True or False)
DAVT	DAVF	Data Valid (True or False)
NDACT	NDACF	Not Data Accepted (True or False)
NRFDT	NRFDF	Not Ready for Data (True or False)
LENB		Listen without ATN Enable
LDIS		Listen without ATN Disable

Bus Traffic:

This depends on the results of the string expression(s):

IFC		One-shot pulse which lasts approximately 150 microseconds
RENT	RENF	The selected IEEE 488 interface line is set true (asserted) or false (unasserted).
EOIT	EOIF	
ATNT	ATNF	
SRQT	SRQF	
DAVT	DAVF	
NDACT	NDACF	
NRFDT	NRFDF	
LENB		No traffic
LDIS		

Application Example:

Fatal errors can leave the low-level IEEE 488 Interface driver (GPI.SPS) in an unknown state in which it may or may not be able to accept subsequent communication. As a result, more errors (usually I18 errors) can be generated during later processing. To avoid this the SIFLIN command can be used to clear the interface after a fatal error. Also, the device being communicated with when the error occurred should be reset. For example:

```
SIFLIN @N,"IFC"
SIFCOM @N,LA,SA,"SDC"
```

clears interface number N and sends a Selective Device Clear to the specified device, where LA and SA are the primary listen address and the secondary address of the last device communicated with.

SIFTO

Examples:

```
880 SIFTO @N,TX
150 SIFTO @1,-1
```

Syntax Form:

[line no.] **SIFTO** @expression,expression

Descriptive Form:

[line no.] **SIFTO** @IEEE 488 interface number, time-out value in milliseconds

Purpose:

To set the interface time-out value.

Discussion:

SIFTO (Set InterFace Time-Out) specifies the amount of time in milliseconds that the driver should wait for a response before "timing out". The interface may time out if no listener responds with the handshake when the interface is attempting to write to the bus, or if no device is attempting to talk on the bus when the controller is a listener. The time-out value (default of 5 milliseconds) is used by GET, PUT, READBINARY, RBYTE, WBYTE, RASCII, WASCII, GETSTA, POLL, and SIFCOM.

NOTE

The interface driver was not designed to provide precise timing with the time-out value. It does, however, take into account the timing characteristics of a Digital Equipment Corp. 11/05, 11/20, 11/34, 11/35, or 11/45 processor with core memeory and a CP4165 (or other DEC LSI-11 based controller) with semiconductor memory.

Using the Command Syntax:

The expression following the at sign (**@**) is the number of the IEEE 488 interface which is assigned the time-out value. The second expression specifies the time-out value in milliseconds. The driver default time is 5 milliseconds. Specifying -1 indicates that the driver should wait indefinitely.

Bus Traffic: NONE

STERMC (Nonresident)

Examples:

```
700 STERMC @N,"^J"
190 STERMC @1,CHR(13)&CHR(10)
340 STERMC @0,A$
```

Syntax Form:

[line no.] **STERMC** @expression,string expression

Descriptive Form:

[line no.] **STERMC** @IEEE 488 interface number, specification of termination character(s)

Purpose:

To designate the termination character(s) for ASCII data which is read into a string variable by GET or RASCII from the IEEE 488 interface.

Discussion:

STERMC (Set TERMination Characters) specifies the character(s) to be recognized as the termination string for ASCII data received if the termination string appears prior to an EOI. It only applies to data input from the IEEE 488 interface by a GET or RASCII command and stored in a string variable. This allows the user to break ASCII data into logical segments. For example, a carriage return and/or line feed may be specified to assign one "sentence" to each string variable.

The terminating character(s) are not stored in the string variable by the GET or RASCII commands. Also, the STERMC command does not affect data transfers into numeric variables or arrays.

Using the Command Syntax:

The expression following the at sign (**@**) is the number of the IEEE 488 interface through which the data transfer is made.

The string expression designates a termination string of zero, one, or two ASCII characters. When it evaluates to one character, inputs into a string variable will terminate on EOI or the first occurrence of the specific character. When two characters are specified, the input terminates on EOI or when the two characters are received consecutively in the specified order. If the string expression evaluates to more than two characters, only the first two are used. When the string expression evaluates to zero characters (the null string), further inputs from the IEEE 488 bus into a string variable **terminate only on EOI**.

Bus Traffic: NONE

TIFL (Nonresident)

Examples:

```
990 TIFL @3,X
110 TIFL @N,X(N)
```

Syntax Form:

[line no.] **TIFL** @expression,variable

Descriptive Form:

[line no.] **TIFL** @IEEE 488 interface number, target variable

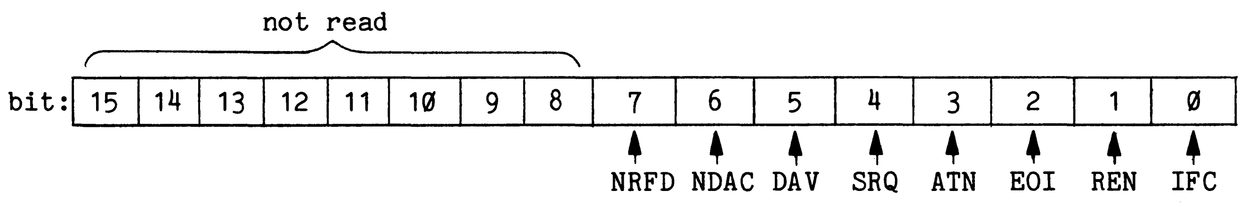
Purpose:

To read the current settings of the control lines of an IEEE 488 bus.

Discussion:

TIFL (Test InterFace Lines) reads the current value of the IEEE 488 control lines and returns their settings in the variable specified.

TIFL reads this information from the eight low-order bits of the Bus Status Register (BSR) of the IEEE 488 interface to which the bus is connected. The data is stored in the register in the following order:



A 1 in the corresponding bit means the line is true (asserted) while a zero means false (unasserted). TIFL converts the binary number represented by this byte to a decimal number and returns it in the specified variable. This information can be decoded with the VARTST command.

For example, to test the DAV line for true on the bus connected to interface number N you could use:

```
TIFL @N,L
VARTST L,"4Ø",C
```

Here the decimal value for the current settings of the control lines are returned in L. Then the VARTST statement tests if bit five is one (true) by comparing L to octal 4Ø (1ØØØØØ binary). If DAV is asserted (true), C will equal 1; if DAC is unasserted (false), C will equal zero.

Bus Traffic: NONE

WASCII (Nonresident)

Examples:

```
150 WASCII X;A$ INTO @0,LA,SA
440 WASCII B$;INTO @N,X;LA,SA
370 WASCII X,Y;A$ INTO @1
990 WASCII S$;A INTO @N,B(0,J)
```

Syntax Form:

```
                     ( expression         ) [ ( , ) ( expression         ) ]
[line no.] WASCII    { array expression   } [ {   } { array expression   } ] ... [;]
                     ( string expression  ) [ ( ; ) ( string expression  ) ]

                     INTO @expression [ , { array expression              } ]
                                      [   { expression[,expression]       } ]

                     [ ; { array expression              } ] ...
                     [   { expression[,expression]       } ]
```

Descriptive Form:

```
                     ( source expression         ) [ ( , ) ( source expression         ) ]
[line no.] WASCII    { source array expression   } [ {   } { source array expression   } ] ... [;]
                     ( source string expression  ) [ ( ; ) ( source string expression  ) ]

                     INTO @IEEE 488 interface number [ , { listen and secondary address pairs   } ]
                                                     [   { listen address[,secondary address]   } ]

                     [ ; { listen and secondary address pairs   } ] ...
                     [   { listen address[,secondary address]   } ]
```

Purpose:

To send ASCII data to bus-connected devices. Numeric data is converted to ASCII before it is sent.

Comments:

WASCII (Write ASCII) sends both strings and numbers to bus-connected devices as ASCII characters. WASCII converts a number to a string of ASCII characters which represent the decimal value of the number and then sends that string. This string is called a numeric string. WASCII uses the same routine to convert a number to a numeric string that the LIST command uses when it outputs a numeric constant in a program listing. This gives you up to seven decimal digits of accuracy.

The default time-out value for addresses that do not respond is five milliseconds. This time-out value can be changed by the SIFTO command.

The bus traffic with WASCII depends on the options selected. If the optional list of addresses is specified, before the data is sent, WASCII unasserts NRFD, asserts REN, and untalks and unlistens all devices on the bus. It then sends the list of addresses with ATN asserted making the specified devices listeners. Finally, after sending the last data byte, WASCII again untalks and unlistens all devices on the bus.

If the address list is omitted, WASCII unasserts NRFD before the data is sent but no interface messages are sent after the data is sent. The IEEE 488 bus is left in the same state that it was in before the execution of WASCII except that NRFD is unasserted. This means that it is the user's responsibility to be sure that there is no other talker and to set up the desired listener(s). (See WBYTE for an example of how to do this using SIFCOM statements.)

An EOI is automatically sent concurrently with the last data byte unless a semicolon is specified following the list of data arguments.

Using the Command Syntax:

The list of arguments preceding the keyword **INTO** specifies the data to be sent. If a numeric expression is specified, the decimal value is converted to an ASCII numeric string before it is sent. If an array is specified, each element is converted to an ASCII numeric string before it is sent. For an array, a comma is sent between each converted element to delimit the numeric strings. If a string expression is specified, the resulting string is sent.

The arguments in the data list may be separated by either a comma or a semicolon. If a comma is used, the ASCII code for a comma is sent. If a semicolon is used, no delimiter is sent.

If the optional semicolon is specified at the end of the list of data arguments, EOI is not sent with the last data byte. If the ending semicolon is omitted, EOI is sent concurrently with the last byte of data.

The expression following the at sign (@) is the number of the IEEE 488 interface through which the data is sent.

The optional list of addresses follows the interface number. The list elements, which may be array expressions or numeric expressions, are separated by semicolons.

If an array expression is specified, the resultant array must contain an even number of elements. The elements must be pairs of addresses: a primary listen address followed by a secondary address. If the primary address is negative, the address pair is not sent. If the secondary address is negative, only the primary address of the pair is sent.

If a numeric expression is used, a comma separates the primary listen address from the optional secondary address. When the address is a numeric expression, specifying negative numbers is illegal.

Expressions for primary listen addresses must evaluate to numbers between 32 and 62 while secondary addresses must be between 96 and 126. An exception to this is if the address is specified in an array expression and a negative value is given to cause WASCII to ignore (skip over) either the address pair or the secondary address, as explained earlier.

Bus Traffic:

[ATN UNT ATN UNL ATN listen address [ATN secondary address]] . . .	If optional list of addresses are specified, the controller untalks and unlistens all devices on the bus and sends the listen address and secon addresses from the list.

<first ASCII data byte> . . .	Data byte sent to listener(s) from source expression(s)
[EOI] <last ASCII data byte>	EOI optionally sent with last data byte
[ATN UNT ATN UNL]	If optional list of addresses are specified; the controller untalks and unlistens all devices on the bus.

Application Example:

WASCII gives you a simple way to send numeric data to devices that only accept ASCII code. For example, you could use WASCII instead of PUT to set the Time/Division of a TEKTRONIX 7B90P Programmable Time Base plug-in. Here the interface number is N and the primary listen and secondary addresses of the plug-in are LA and SA, respectively. So the statements:

```
T$="T/D "
T=5E-Ø3
WASCII T$;T INTO @N,LA,SA
```

would send the ASCII code for the string "T/D 5E-Ø3" to the 7B90P, setting the Time/Division to 5 milliseconds. Notice that since a semicolon separates T$ and T, no comma is sent between T$ and the numeric string representing T.

WBYTE (Nonresident)

Examples:

```
220 WBYTE @3,1
840 WBYTE @N,Y,Z
```

Syntax Form:

$$\text{[line no.] } \textbf{WBYTE} \text{ @expression,} \left\{\begin{matrix}\text{expression}\\ \text{array expression}\end{matrix}\right\} \left[, \left\{\begin{matrix}\text{expression}\\ \text{array expression}\end{matrix}\right\} \right] \ldots \text{ [,]}$$

Descriptive Form:

$$\text{[line no.] } \textbf{WBYTE} \text{ @IEEE 488 interface number,} \left\{\begin{matrix}\text{source expression}\\ \text{source array expression}\end{matrix}\right\}$$
$$\left[, \left\{\begin{matrix}\text{source expression}\\ \text{source array expression}\end{matrix}\right\} \right] \ldots \text{ [,]}$$

Purpose:

To send a byte of data through an IEEE 488 interface to the bus.

Discussion:

WBYTE (Write BYTE) is used to write one or more bytes of data to the bus, one byte at a time. If the listener does not accept a byte of data within the time allotted by the time-out value, an error message is issued. (The driver has a default time-out value of 5 milliseconds. This time-out value can be changed by the SIFTO command.)

Note that before the WBYTE command is executed, the instruments to receive the data must be told to listen via the SIFCOM command. Before the data is sent, WBYTE unasserts NRFD.

Using the Command Syntax:

The expression following the at sign (**@**) is the number of the IEEE 488 interface through which the data is sent.

The data to be sent is specified by the list of expressions following the interface number. A list element may be either a numeric expression or an array expression. For each numeric expression or array element, the modulo 256 value of the integer part of the number is transmitted to the interface.

If the optional comma following the list of expressions is specified, EOI is not sent with the last data byte. Otherwise EOI is automatically sent concurrently with the last byte.

Bus Traffic:

```
          <data byte>                    Data byte sent from source
            .                            expression(s).
            .
            .
  [EOI]   <data byte>                    EOI optionally sent with last
                                         data byte
```

Application Example:

This routine sends the ASCII codes for a string of characters to a bus-connected device through IEEE 488 interface number N. It uses WBYTE in a FOR/NEXT loop to send the data one byte at a time. The routine then sends a carriage return, line feed following the ASCII code for the last character.

```
100 SIFLIN @N,"RENT"
110 SIFCOM @N,"UNT","UNL",LA,SA
120 FOR I=1 TO LEN(B$)
130 WBYTE @N,ASC(SEG(B$,I,I)),
140 NEXT I
150 WBYTE @N,13,10
160 SIFCOM @N,"UNL"
```

First, line 1ØØ asserts REN in case it has not already been asserted by a PUT or WASCII statement. Next, line 11Ø untalks and unlistens all the devices on the bus and makes the selected device a listener by sending its primary listen address (LA) and its secondary address (SA) with ATN asserted. Then lines 12Ø through 14Ø send the ASCII code for each character in B$. The comma at the end of line 13Ø prevents the WBYTE from sending an EOI. Next, line 15Ø sends the ASCII code for a carriage return and a line feed. Since the comma is omitted from the end of line 15Ø, an EOI is asserted with the last byte of data, the ASCII code for a line feed. Finally, line 16Ø unlistens the device.

SECTION 7

GLOSSARY

Algorithm. A fixed, step by step, procedure for accomplishing a given result; usually a simplified procedure for solving a complex problem.

Argument. The user-specified information sent to a function or command.

Array. A series of data storage elements referenced by 1 or 2 indices and a common name.

Array Expression. Any legal combination of constants, variables, waveforms, arrays, arithmetic operators, functions, and parentheses that evaluate to an array.

Array Zone. A contiguous portion of an array.

ASCII. (American Standard Code for Information Interchange) A standard code used to represent letters (A-Z and a-z), digits (Ø-9) and a set of special characters. Most computers support software that recognizes ASCII code.

Assembler. A program that converts symbolic assembly language into machine language. An assembler generally translates assembly language instructions into machine language instructions on a one-for-one basis.

Assembly Language. A machine oriented language whose symbolic statements correspond one-to-one with machine language instructions. TEK SPS BASIC is written in PDP-11 assembly language.

Attached Instrument. An instrument associated with an instrument logical unit number (ILUN) by the ATTACH command.

Auto-Dimension. To dimension a simple numeric variable, without using the DIM command, by setting it equal to an array expression. The resultant array is a one-dimensional array with the same number of elements as the array to which the expression evaluates.

Auto-Load. To bring a nonresident command from the system storage device into resident BASIC by invoking the command itself without using the LOAD command.

Bad Block. A physically damaged or physically unsatisfactory block of memory on a storage device. The ".BAD" extension is used to signal the location of damaged blocks.

Bit. A binary digit, the smallest element of binary machine language. Also the smallest possible unit of information. A bit of information can be a 1 or a Ø. See Byte, Word.

Block. A group of consecutive words which are handled as a single unit especially during I/O operations. In TEK SPS BASIC one block is typically 256 words.

Bootable Device. A peripheral device from which an operating system can be loaded. In TEK SPS BASIC, an absolute loader for the system software must be installed on the medium in the device in order for the system software to be loaded. (An absolute loader can be installed by either HOOK or HOOKQ.) Examples of bootable devices are disk drives such as DK and DX.

Byte. A group of binary digits (bits) usually operated upon as a unit. In dvTEK SPS BASIC a byte is 8 bits long. See Bit, Word.

Compiler. A program that converts a symbolic, machine independent, high-level language such as FORTRAN, COBOL, or PL-1 into computer dependent machine language. A compiler is machine dependent.

Concatenate. To connect or chain together.

Controllers. A term for computers that control instruments or processes.

Data Sampling Interval. The time increment between the data points of a waveform.

Delimiter. A character used to mark the beginning and/or end of a unit of data.

Destination. The place to which data is transferred. The recipient of data.

Device Name. A two or three letter mnemonic that is used when referencing peripheral or instrument devices.

Directory-Structured Device. A peripheral using a file name directory to locate its data files.

Drive Number. A hardware number in the range 0 to 77777 (octal) determined by the hardware strapping configuration.

Driver. A software module that communicates with a peripheral device or an instrument.

DSI. An abbreviation for data sampling interval.

Enable. To make an interrupt possible.

Expression. Any constant, variable, or legal combination of constants, variables, waveforms, arrays, arithmetic operators, functions, and parentheses that evaluates to a single value.

Extended Memory. Up to 96K words of additional memory beyond the standard 28K words of controller memory. In standard memory systems, this memory can be used only as a peripheral storage device accessed by the Virtual Memory driver (VM.SPS). In extended memory (XM) systems, this memory is used for program data (numeric arrays only) and/or as a peripheral device, depending on how the system software is initialized at load time. (See the SYSBLD command.)

Fatal Error. An error so serious that it halts execution of the task in which it occurs.

Field. A specific part of a data record.

File. A collection of related records treated as a unit. TEK SPS BASIC also uses the term to refer to the named (Filnam.Ext) physical area on a mass storage device. By convention, the extension to the file name describes the type of content. For example .BAS is an extension for a basic program and .BIN, for a collection of binary numbers.

File-Structured Device. A peripheral on which data is accessed by a file name.

Firmware. A program or programs that have been committed to read-only memory. The program becomes a permanent part of the computer until the memory is physically removed.

Hardware. The mechanical, magnetic, electrical and electronic devices or components of a system.

Hash Function. A procedure that maps a key (e.g., a name or an identificatio number) to an address for storing and later retrieving the information associated with the key. The HASH command performs a hash function.

HUN. An acronymn used in TEK SPS BASIC to stand for the Hardware Unit Number, which is the physical address strapped into an instrument.

Idle Mode. The state of a computer when it is available for use and waiting for instruction.

ILUN. A TEK SPS BASIC acronymn that stands for Instrument Logical Unit Number. For ease and flexibility of programming, the ILUN is associated with a specific instrument and then is used when communicating with that instrument.

Immediate Mode. The state of a computer in which a command that is not preceded by a line number is executed as soon as it is entered from the terminal and the Return key is pressed.

Interpreter. A program that translates the intent of a program written in a high level language such as BASIC or APL directly into computer actions, on a line-at-a-time basis.

Interrupt. In TEK SPS BASIC, an external signal that can be used to initiate transfer of execution control to a specified line in the BASIC program. It is an external demand for service that is recognized by the computer hardware and software.

I/O. The acronymn for Input/Output. It is loosely applied to any movement of data into or out of the computer from a peripheral or instrument.

Keyword. A word recognized by TEK SPS BASIC. For example: LET, GOTO, IF, FOR, NEXT, etc.

Machine Language. A coding scheme that can be read and executed by the computer without further processing. When printed, machine language usually appears as a sequence of fixed length numbers.

Module. A unit of special purpose software such as a driver or nonresident command.

Nonresident. As used in TEK SPS BASIC, it refers to drivers and commands which which can be individually loaded from a mass storage device. Examples include the GRAPH command, the line printer driver, and the DPO driver.

Null String. A string with no length, that is, "".

Numeric Variable. A symbol, representing a numeric value, whose content may change during the execution of a program or from one execution to the next.

Overflow. A condition in which an arithmetic operation results in a number too large to express.

Overlaying. The technique of using one small piece of computer memory to execute many commands. When a command is needed, it is read from a mass storage device into the designated memory area. The command that previously occupied the memory is overwritten.

Packet. A group of data words treated as a unit. A three-word packet is entered in the Scheduler for each subroutine scheduled. This packet contains the line number (internal address), priority, and task number associated with the subroutine.

Patch File. A file that holds a routine to correct an error or make updates in software.

Pending Routine. A routine that has been entered in the Scheduler queue but has not started to execute.

Peripheral. A device for storage and/or retrieval of data.

PLUN. A TEK SPS BASIC acronymn that stands for Peripheral Logical Unit Number. For ease of programming, the PLUN is associated with a specific device or file and is used when doing I/O with that peripheral.

Priority. The measure of importance attached to a program. When two or more program segments attempt to execute at the same time (this can happen if interrupts are allowed), TEK SPS BASIC executes the program with the highest priority first. Priority is an actual number that can be associated with programs by the programmer using special commands or keywords.

Program. A sequence of steps that can be readily translated to machine language and executed by a computer.

Program Mode. The state of a computer in which commands preceded by line numbers are executed.

RAM (Random Access Memory). An acronymn used to refer to standard read/write memory.

Random Access Storage. A characteristic of a storage device that allows a storage "element" to be read or written without reference to the location of other storage elements. Memory is random access. Disks are usually thought of as random access. For contrast see Sequential Access Storage.

Record. A collection of related and consecutive items of data that is treated as a single unit.

Record I/O File. A file in TEK SPS BASIC that can be accessed randomly, any record at a time. The size of a record is determined by the user.

Resident. As used in TEK SPS BASIC, code that is loaded at initialization time. This code cannot be removed from memory. Examples of resident code are the LET command, the SPS Scheduler, and the system device driver.

ROM (Read Only Memory). Memory that cannot be changed (written) by a program. A program that is committed to ROM is called firmware.

Scheduler. The Resident BASIC software which controls the execution of all routines, based on their priority.

Sequential Access Storage. A characteristic of a storage device that requires storage "elements" to be read or written in "one after the other" sequence. Magnetic tape and paper tape are examples of sequential access storage devices. For contrast, see Random Access Storage.

Serial Tape. A data storage medium that is accessed sequentially.

Simple Variable. A nonsubscripted variable, not an array element.

Software. The loadable program or programs used to direct a computer in its operations.

Solicited Input. Program data that is entered from the keyboard in response to a prompting question mark printed on the terminal by a command such as INPUT.

Source. The place from which data originates.

String. In TEK SPS BASIC, a series of ASCII coded characters that are treated as a logical unit. Strings are stored in string variables, which are designated by a trailing $ (i.e., A$, B$). Strings are manipulated by the string operator (&) and string functions supplied in TEK SPS BASIC.

String Array. An array whose elements are strings.

String Variable. A symbol, representing a string and ending in a $, whose content may change during the execution of a program or from one execution to the next.

Subprogram. A subset of a program.

Subroutine. A subset of a program that can be called repeatedly and/or from different parts of the program. A subroutine is terminated by a RETURN statement.

Subroutine Library. A set of standard subroutines which is kept on file for general use.

Syntax. The rules governing acceptable statement formats in a programming language.

System Device. The peripheral device from which TEK SPS BASIC is loaded. It is (usually) used by the I/O commands and by the operating system when peripheral device is specified. The default peripheral device.

System Driver. A module that communicates with the system device.

System Reset. The set of actions that clears the Scheduler and the Clock queue, cancels the actions of all WHEN statements, and disables any INPREQ (input request) and ONERR conditions. These are the same actions caused by a Control-P.

Target. Another word for destination, the recipient of data.

Task. A task is a subprogram that is distinguished from other subprograms by the task number associated with it.

Task Number. The numeric name assigned either explicitly or by default to a subprogram at the time it is entered in the Scheduler queue.

Text. A collection of ASCII characters that may consist of as few as one character or as many as an entire program.

Time-Out. The failure of a device to respond within the allotted time.

Underflow. A condition in which an arithmetic operation results in a number too small to express.

Unsolicited Input. Program data entered from the keyboard before the data has been requested by a command such as INPUT. Unsolicited input is allowed after an input request has been enabled by an INPREQ statement.

Variable. A symbol whose value may change during the execution of a program or from one execution to the next.

Warning Error. An error that interrupts execution to display a warning error message but does not halt execution. Results of the execution may be unpredictable.

Waveform. An array, data sampling interval variable, horizontal units string variable, and vertical units string variable which have been associated with a variable name (any legal numeric variable name) for manipulation as a single entity.

Waveform Expression. Any legal combination of constants, variables, waveforms, arrays, arithmetic operators, functions, and parentheses that evaluates to a waveform.

Wildcard Specification. An asterisk (*) that is used in place of a specific file name or file name extension. It represents all possible names or all possible extensions.

Word. A group of binary digits (bits) that is treated as a unit by the computer. The word is usually the fundamental element of machine language. Word length is computer dependent. A PDP-11 word = 16 bits = 2 bytes. See Bit, Byte.

Zone. A contiguous portion of an array; a subarray.

SECTION 8

UNDERSTANDING ERRORS

Types of Errors

There are two types of errors possible in TEK SPS BASIC, fatal errors and warning errors. Fatal errors cause a task to halt, while warning errors do not. Since a warning error does not stop execution, it is possible to have both a warning error and a fatal error issued at the same time.

Fatal Errors

A fatal error is one so serious that the command cannot execute and the task associated with that command halts. When such an error is encountered, an error message is issued and all places in the system that hold information about that task are cleared. Specifically, the following actions are taken:

1. The error code (category letter and number) is printed on the terminal. If the command is in program mode, its line number and task number are also printed.

2. Any packets with the same task number as the statement committing the error are removed from both the Scheduler stack and queue. (The functions and parts of the Scheduler are explained in Section 1.)

3. The actions of any WHEN statements with the same task number as the statement committing the error are canceled.

4. Any packets with the same task number as the statement committing the error are removed from the Clock queue.

If the fatal error occurs during the execution of an I/O command, the input or output finishes before control is returned to the Scheduler to determine the next task.

Exceptions. Two errors, however, are fatal to <u>all</u> tasks in the system if either one occurs in any current task. These are 1) an overflow of the

Scheduler stack or 2) an overflow of the Scheduler queue. Either of these errors causes a total reset of the system. That is, regardless of task numbers, the Scheduler stack and queue are cleared, all interrupts enabled by WHEN statements are disabled, and the Clock queue is cleared.

Warning Errors

With a warning error, execution is interrupted to print the error code (category letter and number) on the terminal. If the command is in program mode, its line number and task number are also printed. Execution then continues, but the results may not be reliable.

Printout of error messages can be suppressed by execution of one form of the ONERR command prior to the statement containing the error. ONERR can also be used to replace the system's response to errors with your own error-handling routines. See ONERR documentation in Section 4 for complete details.

Error Categories

There are eight categories of errors. Each category is indicated by a letter. The categories are:

Program Control -- Code C. Any error involving the flow of program execution is included in this category. Examples: nonexistent line number in a GOTO or GOSUB statement, invalid line number, or invalid priority value.

Data -- Code D. Data errors cover problems with illegal types of data. Examples: attempting to change the dimensions of a previously dimensioned array, referencing a nonexistent array, or using a numeric argument as a string.

Evaluation -- Code E. These errors concern illegal arithmetic operations. Also included in this category are errors concerning accuracy limitations of the system. Examples: division by zero, an illegal function argument, or arithmetic overflow or underflow.

Hardware/System -- Code H. Computer hardware errors or general system errors are covered by the H category. Examples: controller time-out error or floating-point hardware malfunction.

Instrument -- Code I. Any error concerning the transfer of data or status between the controller and an instrument falls into this category. Examples: attempting to use an instrument that is not on line, referencing an instrument driver not in memory, or using an illegal hardware unit number.

Operating System -- Code O. Errors in this category include problems encountered by the actual operating system. Examples: Scheduler overflow, insufficient free memory, or an attempt to use graphics or string functions if they were deleted at system initialization time.

Peripheral -- Code P. Any error involving input or output with a peripheral device is covered in this category. Examples: data transmission errors, peripheral hardware errors, or an attempt to WRITE on a file OPEN FOR READ.

Syntax -- Code S. This category covers all command syntax errors encountered during statement entry or execution. Examples: keyword used as a command name, incorrect subscript format, or unmatched FOR/NEXT statement pair.

For each category, several errors are possible. The following is a list of the error code messages generated by TEK SPS BASIC. A brief explanation accompanies each error code.

Program Control Errors

CØ RUN command not in immediate mode. RUN is the only BASIC command that cannot be preceded by a line number.

C1 Attempt to pass control to a nonexistent line number. This occurs in GOTO and GOSUB commands when the specified line number does not exist.

C2 Attempt to overwrite program line being executed. Occurs when OVERLAY reads in a line of text with the same line number as the OVERLAY command.

C3 Program line exceeds 8Ø characters. Additional characters are ignored.

C4 Priority value or task number is less than zero or greater than 126. If you are using an expression to indicate a priority value or task number, the result might be outside the legal range of zero to 126, inclusive.

C5 Concatenated statements are in an illegal order. For example, a statement follows an OLD or immediate mode DELETE ALL statement; or a statement precedes a CHANGE statement.

C6 A line of program text with no line number was read from program file. The line is ignored, and loading continues.

C7 The line number of a subroutine scheduled with a SCHEDULE statement is not in memory. Program execution continues as if the subroutine had not been SCHEDULEd.

C8 Illegal ONERR condition when an error occurs. For example, this error is issued if the array specified in the ONERR statement was deleted before the error occurred.

C9 ONERR RETURN statement encountered when no error has occurred. A user-written error-handling routine was entered but not because of an error condition.

C1Ø An instrument or peripheral driver has been auto-loaded as if it were a nonresident command.

C11 Illegal file contents. For example, this error is issued if the OVLOAD command trys to load a file that was not created by the OVLSAV command.

Data Errors

DØ Illegal data on input. For example, an INPUT command has encountered an illegal character, such as a letter (other than E) where a numeric value was expected.

D1 Number too large or too small. The largest possible number expressible in TEK SPS BASIC is approximately 1.7Ø141E+38; the smallest positive fraction is approximately 2.93874E-39. Numbers too large are set equal to the largest possible number, and values too small are set to zero.

D2 String too long. One of the input commands attempted to read in a string longer than 388 characters. The remainder of the string was ignored.

D3 Source data types do not match destination specifications. This error can occur if the data descriptor on the peripheral device has been altered by a hardware error. The data descriptor describes the type of data (numeric scalar, array, waveform, etc.) about to be read.

D4 Simple string or numeric variable appears with a subscript, or string array is referenced without a subscript.

D5 Arrays or waveforms of different lengths. When two arrays are used in an expression, they must be of the same size, or the zoned portions must be of the same size.

D6 Subscript or zone boundary out of range. This occurs if the subscript or one of the zone specifications is negative or greater than the array dimension. This also occurs in extended memory (XM) systems if you try to dimension a floating-point array to more than 8192 elements or an integer array to more than 16384 elements.

D7 Illegal waveform component. This can occur if one of the strings specified in a WAVEFORM statement has been dimensioned to a string array or the data sampling interval variable has been dimensioned to an array.

D8 Source waveforms do not have identical data sampling intervals or horizontal units. Correct evaluation of expressions containing waveforms cannot be accomplished if either of these conditions is found.

D9 Source items are not all waveforms; or if they are all waveforms, their data sampling interval and units are not identical.

D1Ø Correction tables do not contain data required to perform geometry correction. Correction tables are generated by the INSTALL and MAP commands.

D11 Illegal destination type. For example, this error is issued if you attempt to read in a waveform from a peripheral device with the READU command.

D12 Illegal source type. This is the opposite of the D11 error. One example is an attempt to write out a waveform with the WRITEU command.

D13 Illegal address argument. PUTLOC issues this error if the address specified is odd.

D14 Array or waveform previously dimensioned to a different size. You should delete the array before redimensioning it.

D15 Source data sampling interval (DSI) is too small. The DSI of a waveform is used as a divisor in several Signal Processing Package commands. If this value is incorrect or too small, the result of the operation is meaningless.

D16 Calculated data-record length is too long (too many bytes per record are specified) in a record I/O form of a READU or WRITEU statement.

Evaluation Errors

EØ Power operation performed on number less than or equal to zero. If the value is zero, zero is returned. If the value is negative, the absolute value of the argument is used.

E1 Addition overflow. The largest possible number (approximately 1.7Ø141E+38) with the correct sign is returned.

E2 Multiplication overflow. The largest possible number (approximately 1.7Ø141E+38) with the correct sign is returned.

E3 Division overflow. The largest possible number (approximately 1.7Ø141E+38) with the correct sign is returned.

E4 Floating-point-to-integer conversion overflow. The largest possible integer (32767) with the correct sign is returned.

E5 Double-to-single floating-point conversion overflow. The largest possible number (approximately 1.7Ø141E+38) with correct sign is returned.

E6 Addition underflow. Zero is returned.

E7 Multiplication underflow. Zero is returned.

E8 Division underflow. Zero is returned.

E9 Argument of EXP function is less than or equal to -88.5. Zero is returned.

E1Ø Divide by zero. The largest possible number (approximately 1.7Ø141E+38) with the dividend's sign is returned.

E11 Argument of LOG function is less than or equal to zero. Zero is returned.

E12 Argument of EXP function is greater than 88. Largest possible number (approximately 1.7Ø141E+38) is returned.

E13 Argument of SQR function is less than zero. The square root of the absolute value of the argument is returned.

E14 Underflow in power operation. Zero is returned.

E15 Overflow in power operation. Largest positive number (approximately 1.7Ø141E+38) is returned.

E16 Arthmetic overflow during RFFT inverse transform setup. The largest possible number (approximately 1.7Ø141E+38) with the correct sign is used.

E17 Arithmetic underflow during RFFT inverse transform setup. Zero is used.

E18 Divide by zero during RFFT inverse transform setup. The largest possible number (approximately 1.7Ø141E+38) with the dividend's sign is used.

E19 Arithmetic overflow during RFFT computation. The largest possible number (approximately 1.7Ø141E+38) with the correct sign is used.

E2Ø Arithmetic underflow during RFFT computation. Zero is used.

E21 Divide by zero during RFFT computation. The largest possible number (approximately 1.7Ø141E+38) with the dividend's sign is used.

E22 Arithmetic overflow in recovery computations of CONVL or CORR's direct transform. The largest possible number (approximately 1.7Ø141E+38) with the correct sign is used.

E23 Arithmetic underflow in recovery computations of CONVL or CORR's direct transform. Zero is used.

E24 Divide by zero in recovery computations of CONVL or CORR's direct transform. The largest possible number (approximately 1.7Ø141E+38) with the dividend's sign is used.

E25 Arguments of ATAN2 command are both zero. Zero is returned.

E26 Arithmetic overflow during complex multiplication prior to inverse transform in CONVL or CORR. Largest possible number (approximately 1.7Ø141E+38) with the correct sign is used.

E27 Arithmetic underflow during complex multiplication prior to inverse transform in CONVL or CORR. Zero is used.

E28 Divide by zero during complex multiplication prior to inverse transform in CONVL or CORR. Largest possible number (approximately 1.7Ø141E+38 with the dividend's sign is used.

Hardware/System Errors

HØ Controller bus time-out. This is a hardware error in the system controller. It can also occur if you attempt to read or write to a peripheral device not connected to the system.

H1 Illegal controller instruction encountered. This usually indicates that the system software has been altered. A complete reload should be performed if this error occurs.

H2 Floating-point hardware malfunction.

Instrument Errors

IØ More than one type of device illegally sharing an interrupt vector. Check strapping on the interface cards for correct vector addresses.

I1 Instrument driver is not in memory.

I2 Illegal device number. The hardware unit number (HUN) for an instrument is determined by straps on interface cards in the controller and instrument.

I3 Instrument logical unit number (ILUN) already attached to another instrument. DETACH the ILUN before reATTACHing it.

I4 Instrument is not on line.

I5 Instrument logical unit number (ILUN) is not ATTACHed.

I6 Illegal instrument function. Check the instrument driver manual for correct command strings and functions.

I7 Write or timing error on output to a device on the IEEE 488 interface bus.

I8 Interrupt specified in WHEN command occurred, but the specified line number is not in memory. Program execution continues as if the interrupt did not occur.

I9 First horizontal address not found in data transfer from an R7912 Transient Digitizer.

I1Ø Device and device driver are of different types. Load the proper instrument driver.

I11 Four DPOs already ATTACHed. Four is the maximum number of DPOs that can be ATTACHed at any one time.

I12 DPO bus time-out. This is a malfunction in the DPO hardware.

I13 Device specified for R7912 Transient Digitizer fast data log is not a peripheral that supports fast data logging (e.g., DK and DL).

I14 Instrument logical unit number (ILUN) out of range. An ILUN cannot be negative, or larger than the maximum number set at system load time. This system parameter (default value of eight) can be changed by first executing the SYSBLD command and then rebooting.

I15 The specified instrument is already ATTACHed to a different instrument logical unit number (ILUN).

I16 Reserved error code. It is not used in this version of BASIC.

I17 Interrupt occurred on IEEE 488 interface bus for "ERR", "EOI", or "SRQ", but no interrupt condition exists.

I18 Device on IEEE 488 interface bus did not accept or send data within the time-out period. The time-out value can be changed with the SIFTO command. Also, the device may not be functioning or the interface may need to be cleared with the SIFLIN command.

I19 Insufficient or excessive data available for the variables specified. You may have too many items in the list of arguments, or an array may be dimensioned to too many or too few elements. Also, a packed data transfer mode requires half the array size of an unpacked mode.

I2Ø Checksum error in an IEEE 488 interface bus data transmission. This is probably a hardware error.

I21 Empty binary block transmission received. If the target is a simple numeric variable, it is <u>not</u> autodimensioned to an array. If the target is an array, there is no change to its data. This error may occur, for instance, while trying to acquire the verticals array from a 7912AD when the main intensity is too low.

Operating System Errors

0Ø Scheduler stack overflow. Too many routines have been stacked to allow for other, higher priority routines to execute. This error is fatal to all tasks, not just the task associated with the statement that triggered the error.

01 Scheduler queue overflow. Too many interrupts have occurred or too many routines have been SCHEDULEd. This error is fatal to all tasks, not just the task associated with the statement that triggered the error.

02 Insufficient free memory. Try releasing nonresident commands and drivers and deleting REM statements. Also, you could break your program into smaller segments and use program overlays. If this error occurs while replacing a program file, be sure to free at least 512 words of memory (try deleting an array) and execute the REPLACE or OVLSAV command again. The old version of the program file may have been canceled, but the updated version was not saved.

03 Maximum number of nonresident commands, peripheral drivers, or instrument drivers has already been loaded. The maximum number of these modules that can be in memory at any one time is defined at system load time. (The default values are six nonresident commands and four each peripheral and instrument drivers.) These system parameters can be changed by first executing the SYSBLD command and then rebooting.

04 String functions deleted at load time. To use these functions, you will have to first execute the SYSBLD command to alter the file of user defined initialization parameters and then reboot.

05 Graphics option deleted at load time. To use the graphics command, you will have to execute the SYSBLD command to alter the file of user-defined initialization parameters and then reboot.

06 Auto-load feature not possible from system drive.

07 Temporary strings have been deleted while still in use. This can happen during execution of a user-written command module if certain precautions are not taken when calling for a downpack of the string area.

08 Nonresident command or driver has attempted to move upper memory in order to obtain more room by releasing a buffer or deleting an array. This error would usually occur when a user-written module attempts to move itself in memory. The command is not executed, and the program stops. Deleting arrays in immediate mode and restarting the program at the line where the error occurred should solve the problem.

09 Clock queue overflow. More subroutines have been SCHEDULEd than can be stored in the clock driver. Maximum number of subroutines that can be SCHEDULEd at one time is 24.

01Ø IEEE 488 (GPIB) capabilities deleted at system software load time. If you want to use the IEEE 488 capabilities, you will have to execute the SYSBLD command to change the user-defined parameter in "SYSBLD.DEF" and then reboot.

011 Peripheral or instrument driver name is too long. No more than three characters are allowed before the .SPS extension.

012 Nonresident module is incompatible with version of monitor.

Peripheral Errors

PØ Illegal use of keyboard. The system terminal keyboard may not be explicitly OPENed FOR READ or WRITE; it is always defined as peripheral logical unit number (PLUN) zero.

P1 Peripheral logical unit number (PLUN) not OPEN FOR READ.

P2 Peripheral logical unit number (PLUN) not OPEN FOR WRITE.

P3 Logical end-of-file reached but no transfer of control provided via EOF command.

P4 Command cannot execute because all available peripheral logical unit numbers (PLUNs) are in use. Close a file before executing the command again.

P5 Referenced file already exists on medium. You cannot OPEN a file for WRITE if that file already exists on the same device you are trying to write to.

P6 Illegal use of the system device driver.

P7 Peripheral driver referenced is not in memory.

P8 Specified file is already OPEN.

P9 Specified file does not exist.

P1Ø Driver specified for RELEASE is still in use.

P11 Physical end-of-file reached. There is no more room in the file.

P12 COPY command's source and destination files are the same, or one or both of the specified files are OPEN.

P13 Illegal function for specified driver.

P14 Illegal function with OPEN file, such as ZEROing the media, or canceling an OPEN file.

P15 Peripheral device not ready. Be sure the device is turned on and ready.

P16 Device is full.

P17 Device directory is full. You can specify how many blocks you want to reserve for a directory when you ZERO a device.

P18 Hardware input/output error.

P19 Illegal device number. The drive number for a peripheral device is determined by straps on the interface card or in the device itself.

P2Ø Peripheral logical unit number (PLUN) is out of range. A PLUN cannot be negative or larger than the maximum number set at system load time. This system parameter (default value of six) can be changed by first executing the SYSBLD command and then rebooting.

P21 Physical bounds of flexible disk exceeded. ZERO the disk before reusing it if error occurred with SQUISH command.

P22 Unrecognized input/output media format.

P23 Device not currently addressable.

P24 Peripheral logical unit number (PLUN) specified in the record I/O form of a READU or WRITEU command is not OPEN FOR UPDATE.

Syntax Errors

SØ   Illegal command name. A keyword was used in place of a command name.

S1   Illegal character in source statement.

S2   Illegal item within parentheses or parentheses unmatched. Be sure you have the same number of closing parentheses as opening parentheses.

S3   Operator or argument omitted in statement.

S4   Illegal array zone use or illegal colon. Only one subscript of a two-dimensional array may be zoned.

S5   Illegal function argument. Check the type of data the function expects to see.

S6   Illegal driver specification.

S7   No space after keyword. This is the only place where a space is required in BASIC.

S8   Line number incorrectly used, omitted, or out of range. The largest possible line number is 32767.

S9   Illegal numeric argument. The command does not allow the specified argument, such as a numeric or string variable where an array is required.

S1Ø Illegal or missing delimiter. Might be a comma out of place, or a backslash (end of command) in the wrong location. This error is also issued if a required argument is not present in the statement.

S11 Variable, array, subarray, waveform, or string variable not found where expected.

S12 Missing keyword or keyword not found where expected.

S13 Unmatched FOR/NEXT statements. Each FOR statement needs a matching NEXT statement. FOR/NEXT loops may not partially overlap.

S14 Illegal subscript or zone specification. Check to see that the subscript value specified is in the range of zero to the maximum index of the array.

S15 Illegal file name. File names must not contain any character other than letters or digits.

S16 No equal sign where expected.

S17 LET source type does not match destination type. This might be caused by assigning a numeric value to a string, or by an improper use of an array or waveform.

S18 Illegal item in expression.

S19 Illegal or missing relational operator in IF statement.

S2Ø Argument type does not match operator, such as concatenating two variables, or adding two strings.

S21 Illegal string function argument. Make sure that the argument specified matches the type expected by the function. Some string functions require numeric values, others require strings.

S22 Illegal or missing command argument.

S23 Too many parentheses in expression. The maximum number of parentheses allowed depends on the complexity of the statement.

APPENDIX A

ASCII & IEEE 488 (GPIB) CODE CHART

BITS B7 B6 B5 →	Ø Ø Ø	Ø Ø 1	Ø 1 Ø	Ø 1 1	1 Ø Ø	1 Ø 1	1 1 Ø	1 1 1
B4 B3 B2 B1	CONTROL		NUMBERS SYMBOLS		UPPER CASE		LOWER	
Ø Ø Ø Ø	0 NUL 0 (0)	20 DLE 10 (16)	40 SP 20 (32)	60 0 30 (48)	100 @ 40 (64)	120 P 50 (80)	140 ` 60 (96)	160 p 70 (112)
Ø Ø Ø 1	1 GTL SOH 1 (1)	21 LLO DC1 11 (17)	41 ! 21 (33)	61 1 31 (49)	101 A 41 (65)	121 Q 51 (81)	141 a 61 (97)	161 q 71 (113)
Ø Ø 1 Ø	2 STX 2 (2)	22 DC2 12 (18)	42 " 22 (34)	62 2 32 (50)	102 B 42 (66)	122 R 52 (82)	142 b 62 (98)	162 r 72 (114)
Ø Ø 1 1	3 ETX 3 (3)	23 DC3 13 (19)	43 # 23 (35)	63 3 33 (51)	103 C 43 (67)	123 S 53 (83)	143 c 63 (99)	163 s 73 (115)
Ø 1 Ø Ø	4 SDC EOT 4 (4)	24 DCL DC4 14 (20)	44 $ 24 (36)	64 4 34 (52)	104 D 44 (68)	124 T 54 (84)	144 d 64 (100)	164 t 74 (116)
Ø 1 Ø 1	5 PPC ENQ 5 (5)	25 PPU NAK 15 (21)	45 % 25 (37)	65 5 35 (53)	105 E 45 (69)	125 U 55 (85)	145 e 65 (101)	165 u 75 (117)
Ø 1 1 Ø	6 ACK 6 (6)	26 SYN 16 (22)	46 & 26 (38)	66 6 36 (54)	106 F 46 (70)	126 V 56 (86)	146 f 66 (102)	166 v 76 (118)
Ø 1 1 1	7 BEL 7 (7)	27 ETB 17 (23)	47 ' 27 (39)	67 7 37 (55)	107 G 47 (71)	127 W 57 (87)	147 g 67 (103)	167 w 77 (119)
1 Ø Ø Ø	10 GET BS 8 (8)	30 SPE CAN 18 (24)	50 (28 (40)	70 8 38 (56)	110 H 48 (72)	130 X 58 (88)	150 h 68 (104)	170 x 78 (120)
1 Ø Ø 1	11 TCT HT 9 (9)	31 SPD EM 19 (25)	51) 29 (41)	71 9 39 (57)	111 I 49 (73)	131 Y 59 (89)	151 i 69 (105)	171 y 79 (121)
1 Ø 1 Ø	12 LF A (10)	32 SUB 1A (26)	52 * 2A (42)	72 : 3A (58)	112 J 4A (74)	132 Z 5A (90)	152 j 6A (106)	172 z 7A (122)
1 Ø 1 1	13 VT B (11)	33 ESC 1B (27)	53 + 2B (43)	73 ; 3B (59)	113 K 4B (75)	133 [5B (91)	153 k 6B (107)	173 { 7B (123)
1 1 Ø Ø	14 FF C (12)	34 FS 1C (28)	54 , 2C (44)	74 < 3C (60)	114 L 4C (76)	134 \ 5C (92)	154 l 6C (108)	174 ¦ 7C (124)
1 1 Ø 1	15 CR D (13)	35 GS 1D (29)	55 – 2D (45)	75 = 3D (61)	115 M 4D (77)	135] 5D (93)	155 m 6D (109)	175 } 7D (125)
1 1 1 Ø	16 SO E (14)	36 RS 1E (30)	56 . 2E (46)	76 > 3E (62)	116 N 4E (78)	136 ^ 5E (94)	156 n 6E (110)	176 ~ 7E (126)
1 1 1 1	17 SI F (15)	37 US 1F (31)	57 / 2F (47)	77 UNL ? 3F (63)	117 O 4F (79)	137 UNT _ 5F (95)	157 o 6F (111)	177 RUBOUT (DEL) 7F (127)

ADDRESSED COMMANDS

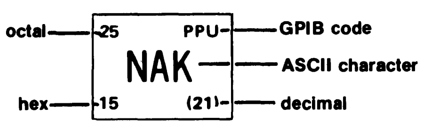
UNIVERSAL COMMANDS

LISTEN ADDRESSES

TALK ADDRESSES

SECONDARY ADDRESSES OR COMMANDS

KEY TO CHART



APPENDIX B

ARCHIVING YOUR SOFTWARE

We strongly urge you to create working copies of your software as soon as possible and to keep your original copy as an archive. We also recommend that you never write on your archive-copy medium and if possible, that you always write-protect your archive software when making working copies.

To assist you in archiving your software, some methods for creating working copies are discussed here. The instructions are grouped under two general headings: hard-disk based systems and floppy-disk based systems. Under the heading for your system, find the example that best describes your new software and follow the instructions on how to archive it. The examples include:

Hard-Disk Based Systems

1. System software (without instrument checkout software) on hard disk.

2. System software with instrument checkout software on hard disk.

3. Separate package or module on hard disk.

4. Separate package or module on floppy disk.

5. Instrument checkout software on floppy disk.

Floppy-Disk Based Systems

1. System software on a single floppy disk.

2. TEK SPS BASIC on minimum number of floppy disks.

3. Separate package or module on floppy disk.

4. Instrument checkout software on floppy disk.

After you have made a copy of your software, check to see if it requires patching. Do this by looking in the issue of the SPS Programming Update shipped with your software. Included in this publication are all the reported software errors and patches. Look through the list of patches and the descriptions of the errors they fix. If any of the patches for your version and release of the software are ones you want to implement, carefully follow the patching directions in the SPS Programming Update. Patch the working copy of the software you just made. **Do not patch the archive software.** When you have finished copying and patching your software, store the issue of the SPS Programming Update with your archive software.

NOTE

If you did not receive an issue of the SPS Programming Update with your software, in the U.S.A. request one by writing:

SPS Programming Update
Group 157 (94-384)
Tektronix, Inc.
P.O. Box 500
Beaverton, OR 97077

Outside the U.S.A., contact your local Tektronix representative.

If you do any patching, you may want to save a copy of the patch files created when PATCH.BLD was run. (Such a file has a numeric file name extension, e.g. "PATCH.ØØ1".) **Do not copy these files onto your archive medium.** Instead, copy these patch files onto a separate disk or tape and store this separate medium with the archive software.

Hard-Disk Based Systems

In the discussions that follow, the example device is the DK type of hard disk (a DEC RK05 or similar device). To make working copies of the software on another type of hard disk supported by TEK SPS BASIC, substitute the proper device name for all occurrences of the DK device name, shown in bold.

System Software (without Instrument Checkout Software) on Hard Disk

If you have purchased TEK SPS BASIC system software on hard disk, you have two ways to make a working copy of your software.

I. **SQUISH to a blank disk.** This method is simple but transfers more files to your working copy than you need for a hard-disk system.

1. Load the original disk into drive Ø, bootstrap, and **write-protect**.

2. Load a blank, formatted disk into drive 1.

3. When the system is loaded and READY is printed on the terminal, type:

```
SQUISH DK: TO DK1:
HOOK DK1:
```

4. When READY is printed on the terminal, remove the original disk from drive Ø. This should be stored as the archive copy.

5. The disk in drive 1 is now your working copy of TEK SPS BASIC.

II. **COPY selected files.** With this method, you transfer only the files you need for a hard-disk system onto your working copy.

1. Follow steps 1 and 2 in I above.

2. When the system is loaded and READY is printed on the terminal, type:

```
ZERO DK1:
COPY "*.SPS" TO DK1:"*.SPS"
COPY "*.OVL" TO DK1:"*.OVL"
COPY "PATCH.*" TO DK1:"PATCH.*"
COPY "SPSDK.LDA" TO DK1:"SPSDK.LDA"
HOOK DK1:
```

3. Follow steps 4 and 5 in **I** above.

System Software with Instrument Checkout Software on Hard Disk

If you have purchased TEK SPS BASIC system software with instrument checkout software on a single hard disk, use one of these procedures to create working copies of this software. The type of the checkout software determines which method you use.

I. BASIC checkout software. Use this procedure if the checkout software is a BASIC program (e.g., CP56008 7912AD Checkout Software).

1. Load the original disk into drive Ø, bootstrap, and **write-protect.**

2. Load a blank, formatted disk into drive 1.

3. When the system is loaded and READY is printed on the terminal, type:

```
SQUISH DK: TO DK1:
HOOK DK1:
```

4. When READY is printed on the terminal, remove the original disk from drive Ø. This should be stored as the archive copy.

5. The disk in drive 1 is now your working copy of TEK SPS BASIC with the instrument checkout software.

II. Stand-alone checkout software. Use this procedure if the checkout software is a stand-alone software with its own .LDA file (e.g., CP56001 P7001/R7912 Checkout Software).

1. Follow steps 1 and 2 in **I** above.

2. When the system is loaded and READY is printed on the terminal, type:

```
SQUISH DK: TO DK1:
HOOKQ DK1:
```

3. Follow steps 4 and 5 in **I** above.

When the working disk is bootstrapped, a prompt (*) will appear on the terminal. Any file with the .LDA extension can then be loaded by entering the file name without the file name extension.

To load and execute TEK SPS BASIC, type:

```
SPSDK
```

To load and execute the instrument checkout software, type the name of the .LDA file, but without the .LDA extension.

Separate Package or Module on Hard Disk

If you have purchased a separate software package or supplemental module to add to your system, you have two options when making a working copy.

I. SQUISH to a blank disk. Maintain a separate disk as a working copy of the package or module.

1. Boot TEK SPS BASIC from drive Ø. When the system is loaded and READY is printed on the terminal, type:

```
LOAD "SQUISH"
```

2. Remove the disk with TEK SPS BASIC from drive Ø.

3. Load the original disk with the package or module into drive Ø and **write-protect.**

4. Load a blank, formatted disk into drive 1.

5. Type:

```
SQUISH DK: TO DK1:
```

6. When READY is printed on the terminal, remove the original disk from drive Ø. This should be stored as the archive copy.

7. Use the disk in drive 1 as the working copy of the package or module.

II. COPY to working disk. Add the package or module to your working copy of TEK SPS BASIC on hard disk.

1. Load your working copy of TEK SPS BASIC into drive Ø, bootstrap, but do not write-protect.

2. Load the original disk with the package or module into drive 1 and **write-protect.**

3. When the system is loaded and READY is printed on the terminal, type:

```
COPY DK1:"*.SPS" TO "*.SPS"
```

4. When READY is printed on the terminal, remove the original disk from drive 1. This should be stored as the archive copy.

5. Your working copy in drive Ø now includes the new package or module.

Separate Package or Module on Floppy Disk

If you have purchased a separate software package or supplemental module on floppy disk to add to your hard-disk system, you must have a floppy-disk device. We assume here that you want to add the package or module to your working copy of TEK SPS BASIC on hard disk.

1. Load your working copy of TEK SPS BASIC into hard-disk drive Ø, bootstrap, but do not write-protect.

2. Load the original disk with the package or module into floppy-disk drive Ø and **write-protect** (if possible).

3. When the system is loaded and READY is printed on the terminal, type:

```
LOAD "DX"
COPY DX:"*.SPS" TO DK:"*.SPS"
```

4. When READY is printed on the terminal, remove the original disk from floppy-disk drive Ø. This should be stored as the archive copy.

5. Your working copy in hard-disk drive Ø now includes the new package or module.

Instrument Checkout Software on Floppy Disk

If you received instrument checkout software on a floppy disk, use one of these procedures to add it to your hard-disk system. The type of the checkout software determines which method you use. To transfer the software to your working copy on hard disk, you must have a floppy disk device.

I. **BASIC checkout software.** Follow this procedure to copy checkout software that is a BASIC program (e.g., CP56008 7912AD Checkout Software).

1. Load your working copy of TEK SPS BASIC into hard-disk drive Ø, bootstrap, but do not write-protect.

2. Load the original disk with the checkout software into floppy-disk drive Ø and **write-protect** (if possible).

3. When the system is loaded and READY is printed on the terminal, type:

```
LOAD "DX"
COPY DX:"*.*" TO DK:"*.*"
```

4. When READY is printed on the terminal, remove the original disk from floppy-disk drive Ø. This should be stored as the archive copy.

5. Your working copy in hard-disk drive Ø now includes the checkout software.

II. Stand-alone checkout software. Follow this procedure to copy checkout software that is stand-alone software with its own .LDA file (e.g., CP56001 P7001/R7912 Checkout Software).

1. Follow steps 1 through 3 in I above.

2. Then enter:

```
HOOKQ DK:
```

3. Follow steps 4 and 5 in I above.

Now when the working disk is bootstrapped, a prompt (*) will appear on the terminal. Any file with an .LDA extension can then be loaded by entering the file name, without the file name extension.

To load and execute TEK SPS BASIC, type:

```
SPSDK
```

To load and execute the instrument checkout software, type the name of the .LDA file, but without the .LDA extension.

Floppy-Disk Based Systems

In the discussions that follow, the example device is the DX type of floppy disk (a preformatted, IBM-compatible, single-density flexible diskette). To make working copies of the software on another type of floppy disk supported by TEK SPS BASIC, substitute the proper device name, shown in bold.

System Software on a Single FLoppy Disk

If you have purchased TEK SPS BASIC system software on floppy disk, you have two ways to make a working copy of your software.

I. SQUISH to a blank disk. This method is simpler but transfers more files to your working copy than you need for a floppy-disk system.

1. Load the original disk into drive Ø, bootstrap, and **write-protect** (if possible).

2. Load a blank, formatted floppy disk into drive 1.

3. When the system is loaded and READY is printed on the terminal, type:

```
          SQUISH DX: TO DX1:
          HOOK DX1:
```

4. When READY is printed on the terminal, remove the original disk from drive Ø. This should be stored as the archive copy.

5. The disk in drive 1 is now your working copy of TEK SPS BASIC.

II. Copy selected files. With this method, you transfer only the files you need for a floppy-disk system onto your working copy.

1. Follow steps 1 and 2 in I above.

2. When the system is loaded and READY is printed on the terminal, type:

```
ZERO DX1:6
COPY "*.SPS" TO DX1:"*.SPS"
COPY "*.OVL" TO DX1:"*.OVL"
COPY "PATCH.*" TO DX1:"PATCH.*"
COPY "SPSDX.LDA" TO DX1:"SPSDX.LDA" INTO 95
HOOK DX1:
```

3. Follow steps 4 and 5 in I above.

TEK SPS BASIC on Minimum Number of Floppy Disks

1. Load the original disk with the proper monitor file (SPS**DX**.LDA) into drive 0, bootstrap, and write protect (if possible.)

2. Load a blank, formatted disk into drive 1.

3. When the system is loaded and READY is printed on the terminal, type:

```
LOAD "SQUISH"
SQUISH DX: TO DX1:
HOOK DX1:
```

4. When READY is printed on the terminal, remove the original disk from drive 0. This should be stored as the archive copy.

5. Remove the disk from drive 1. It is now your working copy of TEK SPS BASIC System software.

6. Load another of the original disks to be archived into drive 0 and **write-protect** (if possible).

7. Load another blank, formatted disk into drive 1.

8. Type:

```
SQUISH DX: TO DX1:
```

9. When READY is printed on the terminal, remove the original disk from drive 0. This should be stored as the archive copy.

10. Remove the disk from drive 1. It is now your working copy of additional SPS software.

11. Repeat steps 6 through 10 until all the original disks are archived.

Separate Package or Module on Floppy Disk

If you have purchased a separate software package or supplemental module to add to your system, your have two options when making a working copy.

I. SQUISH to a blank disk. Maintain a separate disk as a working copy of the package or module.

1. Boot TEK SPS BASIC from drive 1. When the system is loaded and READY is printed on the terminal, type:

```
LOAD "SQUISH"
```

2. Remove the disk with TEK SPS BASIC from drive Ø.

3. Load the original disk with the package or module into drive Ø and **write-protect** (if possible).

4. Load a blank, formatted disk into drive 1.

5. Type:

SQUISH **DX**: TO **DX**1:

6. When READY is printed on the terminal, remove the original disk from drive Ø. This should be stored as the archive copy.

7. Use the disk in drive 1 as the working copy of the package or module.

II. Copy to working disk. Add the package or module to your working copy of TEK SPS BASIC on floppy disk. Depending on the number of free blocks on your working disk, you may not be able to do this.

1. Load your working copy of TEK SPS BASIC into drive Ø, bootstrap, but do not write-protect.

2. Load the original disk with the package or module into drive 1 and **write-protect** (if possible).

3. When the system is loaded and READY is printed on the terminal, type:

```
COPY DX1:"*.SPS" TO "*.SPS"
```

4. When READY is printed on the terminal, remove the original disk from drive 1. This should be stored as the archive copy.

5. Your working copy in drive Ø now includes the new package or module.

Instrument Checkout Software on Floppy Disk

Use one of these procedures to archive instrument checkout software. The type of the checkout software determines which method you use.

I. BASIC checkout software. Follow this procedure if the instrument checkout software is a BASIC program (e.g., CP56008 7912AD Checkout Software).

1. Load a blank, formatted disk into drive 1.

2. Boot TEK SPS BASIC from drive Ø. When the system is loaded and READY is printed on the terminal, type:

```
LOAD "SQUISH"
```

3. Remove the disk with TEK SPS BASIC from drive Ø.

4. Load the original disk with the checkout software into drive Ø and **write-protect** (if possible).

5. Type:

```
SQUISH DX: TO DX1:
```

6. When READY is printed on the terminal, remove the original disk from drive Ø. This should be stored as the archive copy.

7. Use the disk in drive 1 as the working copy of the instrument checkout software.

II. Stand-alone checkout software. Follow this procedure if the instrument checkout software is a stand-alone software with its own .LDA file (e.g., CP56001 P7001/R7912 Checkout Software).

1. Follow steps 1 and 2 in **I** above.

2. Enter:

   HOOKQ **DX**1:

3. Follow steps 3 through 7 in **I** above.

When the working disk is bootstrapped, a prompt (*) will appear on the terminal. Any file with an .LDA extension can then be loaded by entering the file name, without the file name extension.

To load and execute TEK SPS BASIC, type:

   SPS**DX**

To load and execute the instrument checkout software, type the name of the .LDA file, but without the .LDA extension.

APPENDIX C

POWER FAIL RECOVERY

TEK SPS BASIC provides protection of data in the event of a power failure. In most cases, pressing the CONTINUE switch on the controller after power is restored is all that is necessary to resume program execution. In some cases, however, the controller may have been in the process of transferring data to or from an instrument or peripheral storage device. In this case, the integrity of the data cannot be assured.

When a power failure is detected by the controller, information about the current state of the system is saved and the processor halts. When power is restored, the following steps are taken:

1. The software performs a timing loop, waiting long enough for the terminal to warm up. (This loop is performed regardless of the type of terminal you have.)

2. The page is erased (a Control-Shift-K, Control-L sequence is sent to the terminal).

3. The message PF @nnnnn is printed at the terminal, and the terminal bell is rung. The line number of the command being executed when the power fail occurred is represented by nnnnn. This value is zero if no program was running when the power failed.

4. The system halts.

At this point you have a choice of options. Pressing CONTINUE on the controller's front panel causes program execution to resume at the point where the failure occurred.

Loading address zero into the front panel switches (all switches down) and pressing LOAD ADDRESS and START causes a complete software reset to be performed. This could cause a system crash if certain non-interruptable processes were occurring when power failed (such as a nonresident command being moved in memory). In this case, a complete software reload is necessary.

It is possible to dump the contents of memory to a peripheral device (such as a disk) after a power failure. After the power fail message is printed and the system halts, the top of the controller's stack contains the address of the first free memory location. A short dump program can be toggled into memory starting at this location to dump memory onto the peripheral.

After a power fail, the address of the top of the stack is contained in memory location 42_8. The stack itself (starting at the address in 42_8) contains the following information:

LOFREE	Address of the lowest free memory location.
HIFREE	Address of the highest free memory location.
R5	Register 5
R4	Register 4
R3	Register 3
R2	Register 2
R1	Register 1
RØ	Register Ø
TPS	Terminal printer status word.
TKS	Terminal keyboard status word.
PC	Program Counter
PS	Processor Status word

Dump Program

The following program can be entered starting at the location specified in LOFREE. This sample program dumps all of memory to hard disk drive zero. Any other peripheral could be used, however. The dump starts at block $2ØØØ_{(8)}$ on the disk. Storage space required for a 28K memory is about $112_{(1Ø)}$ blocks. With this program blocks $2ØØØ_{(8)}$ to 2157 are used.

```
Address   Code      Instruction             Comments

000000    012701    MOV #177412,R1          ;pick up address of
000002    177412                            ;disk address status word
000004    012711    MOV #2524,@R1           ;put in the disk address
000006    002524
000010    005041    CLR -(R1)               ;start dump at address zero
000012    012741    MOV #-160000/2,-(R1)    ;dump all of memory
000014    010000
000016    005741    TST -(R1)               ;point to the status word
000020    105711 X: TSTB @R1                ;wait for disk to be ready
000022    100376    BPL X                   ;loop until ready
000024    012711    MOV #3, @R1             ;set write and go bits
000026    000003
000030    105711 Y: TSTB @R1                ;is it done?
000032    100376    BPL Y                   ;no, wait it out
000034    000000    HALT                    ;and quit when done
```

APPENDIX D

SOFTWARE PATCHING

TEK SPS BASIC gives you the ability to make changes to Resident BASIC, nonresident commands, and drivers. These changes are made when modifications to the software are released by Tektronix via an issue of the SPS Programming Update.

Included in this publication are all the reported software errors and patches. Look through the list of patches and the descriptions of the errors they fix. If any of the patches for your version and release of the software are ones you want to implement, carefully follow the patching directions in the SPS Programming Update. Patch your working copy of the software. **Do not patch the archive software.**

NOTE

If you did not receive an issue of the SPS Programming Update with your software, in the U.S.A. request one by writing:

SPS Programming Update
Group 157 (94-384)
Tektronix, Inc.
P.O. Box 500
Beaverton, OR 97077

Outside the U.S.A., contact your local Tektronix representative.

To help with the modifications, three BASIC programs are included with your software. These programs are:

PATCH.BLD	Creates a data file on the system device which is used by the following programs. Input to this program is supplied by Tektronix in an SPS Programming Update.
PATCH.FIX	Performs the actual patch when Resident BASIC is being modified. This program is automatically executed at system generation time (when you boot

	the system) if you create a user-defined parameter file ("SYSBLD.DEF") by executing SYSBLD and supplying a nonzero value in response to the SYSBLD question "How many words do you want as a patch area?"
PATCH.NRS	Performs the actual patch when a nonresident command or driver is being modified.

If you do any patching, you may want to save a copy of the patch files created when PATCH.BLD was run. (Such a file has a numeric file name extension, e.g., "PATCH.ØØ1".) **Do not copy these files onto your archive medium.** Instead, copy these patch files onto a separate disk or tape and store this separate medium with the archive software.

Resident BASIC Patches

Detailed instructions for creating patch files are supplied along with the patch data from Tektronix. The patch file itself is created by running the BASIC program PATCH.BLD. Input to this program consists of the file name, file length, and the actual file data. As an aid to entering the file contents, PATCH.BLD prompts you with item numbers that match item numbers supplied with the patch data.

The last item entered to the program is a checksum. This value is compared with a checksum computed by PATCH.BLD. If these values do not match, the program stops and you must reenter the data. If the checksums are the same, the newly created patch file is written out to the system storage device, and you are instructed how to edit PATCH.FIX which actually implements the patch.

Once PATCH.FIX has been edited and replaced on the system storage device, the patch is ready to implement. Execute the SYSBLD command to create the user-defined system parameter file "SYSBLD.DEF". When SYSBLD asks the patch size question, enter the value supplied in the SPS Programming Update. If more than one resident patch is to be implemented, add up the number of words required for each and enter this total. After SYSBLD finishes, reboot the system. Since the patch area size is nonzero, the initialization routine will automatically load and run PATCH.FIX which implements the correction.

Patches to Resident BASIC do not change any data stored on the system storage device. Instead, the code in controller memory (after Resident BASIC has been loaded) is altered. This happens each time you boot the system.

Patches to Nonresident Commands or Drivers

When a patch is required in a nonresident command or driver, create the patch file by running PATCH.BLD the same as you would for a resident patch. Then execute PATCH.NRS instead of PATCH.FIX. PATCH.NRS reads the name of the command or driver to be patched, makes the changes, and writes the edited file on the storage device. The original file is saved, with the file name extension changed from .SPS to .BAK. Do not destroy the .BAK file.

Since PATCH.NRS actually changes the code in the nonresident file, it is not necessary to reserve a patch area in memory. Therefore, for a nonresident patch you do not need to create or modify the user-defined parameter file "SYSBLD.DEF".

APPENDIX E

DATA DESCRIPTORS

When you output data to a file with the WRITE command, TEK SPS BASIC inserts data descriptors along with your data. These descriptors are used by the READ command to interpret what kind of data is to be read in. (When data logging, the GET command also inserts data descriptors into the destination file.)

Normally, these descriptors are transparent to you. That is, you can WRITE and READ files without having any knowledge of what the descriptors actually are. However, if you need to read information output by a TEK SPS BASIC WRITE command with software other than TEK SPS BASIC, you need the following information.

Descriptor Format

Data descriptors are one or three bytes in length, and immediately precede the data elements they describe. The descriptors and their meaning can be found in Table E-1.

To better understand how these descriptors are used, let's look at an example of a WRITE command in action. Consider the statement

```
WRITE #1,A,A$,B
```

where A is a three-element floating-point array, A$ contains the string "HI", and B is a floating-point number. Figure E-1 illustrates how this information is written to the file.

TABLE E-1

TYPES OF DATA DESCRIPTORS

Type	**Descriptor**	**Data Element**
Floating-Point Number	Length: 1 byte Byte 1: 373	32-bit floating-point binary number.
Floating-Point Array	Length: 3 bytes Byte 1: 374 Bytes 2-3: Number of elements in array.	32-bit floating-point binary number.
String	Length: 3 bytes Byte 1: 371 Byte 2-3: Number of characters in string.	8-bit ASCII character.
Integer Array (data-logging only)	Length: 3 bytes Byte 1: 375 Byte 2-3: Number of elements in array.	16-bit binary integer.
Array Terminator	Length: 1 byte Byte 1: 372	none
End of Buffer (data-logging only)	Length: 1 byte Byte 1: 376	none
No-op (Data-logging only)	Length: 1 byte Byte 1: 37Ø	none

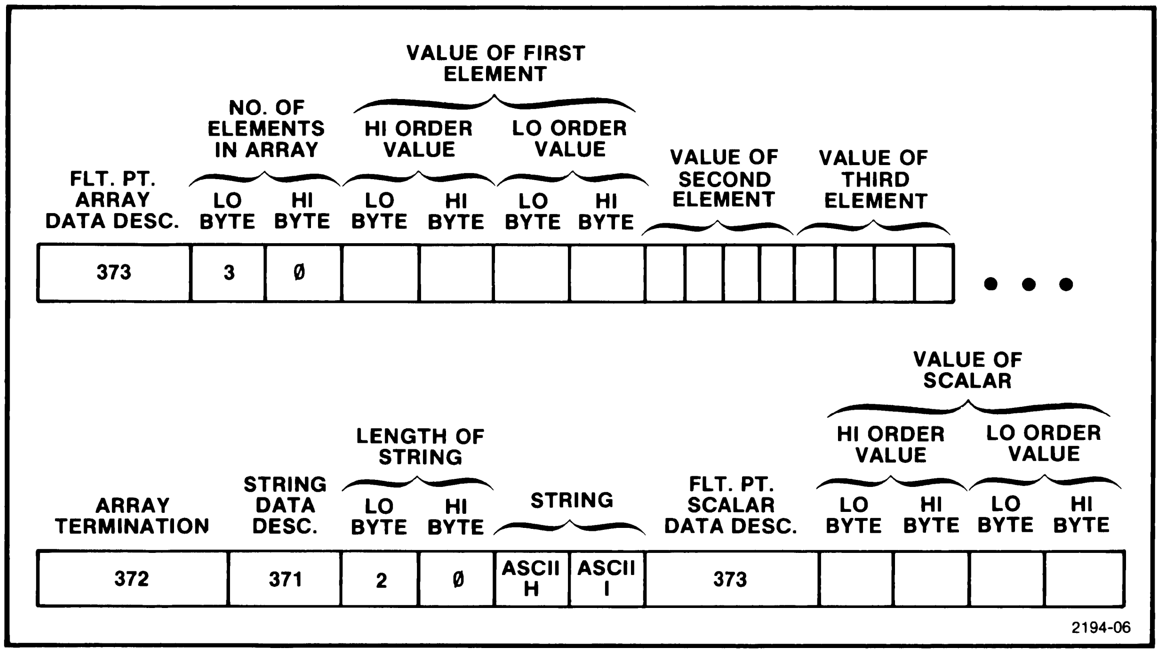


Fig. E-1. Data descriptors as output by WRITE command.

The other way data descriptors are output is by data-logging from an instrument directly to a peripheral file. The following statement shows how this is done.

```
GET #1 FROM #2
```

In this example, peripheral logical unit number one is the destination file on a disk and instrument logical unit number two is the source instrument. Assume ILUN #2 produces an integer array of 1Ø5Ø elements. The output file would look like that described in Fig. E-2.

The NO-OP data descriptor is used in data-logging to put the integer array elements on word boundaries (array output in data-logging is always integer arrays). This allows the data to be placed in the output buffer a word at a time, speeding execution.

The length information refers to the number of elements in this buffer. Since some instruments put out varied length arrays, the GET command has no way of knowing how many elements are to be written until the last element is read. Therefore, the length information is either 254 (the length of an output buffer minus the header information), or the number of elements in the last buffer.

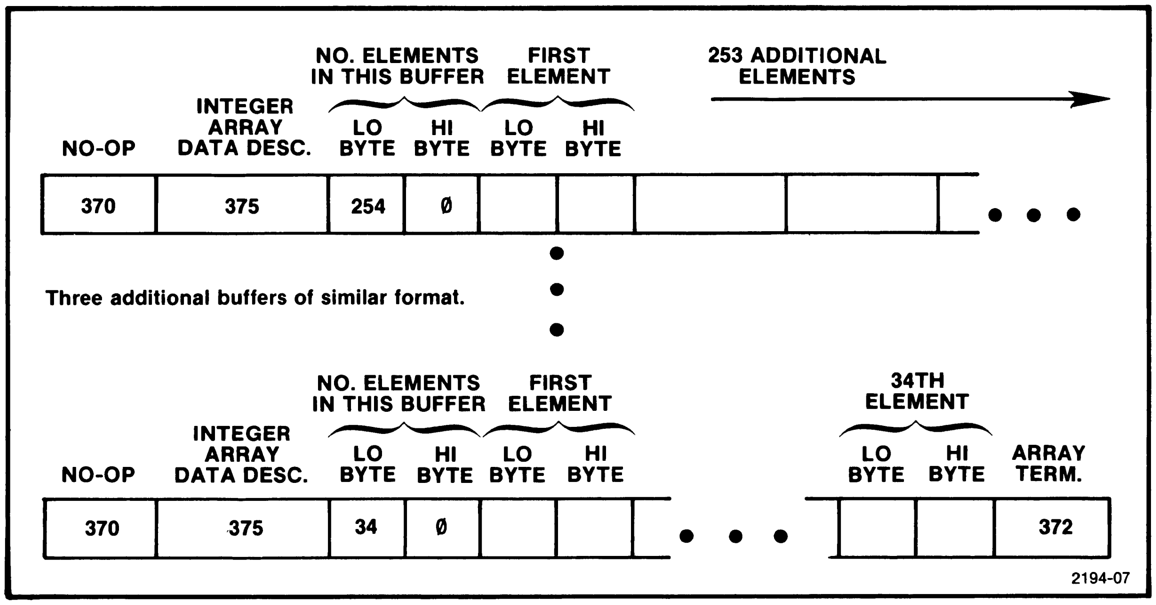


Fig. E-2. Data descriptors as output by GET command when data logging.

In our example of a 1Ø5Ø word array, there are four 254-word buffers and one 34-word buffer. The last element in the last buffer is followed by the array terminator data descriptor.

It is possible, as in the case of 1Ø16-word array, that the last element of the array exactly fills the last buffer. In this case, another buffer with length zero is written to the peripheral file, and the only information in the buffer is the array terminator data descriptor.

The size of the output buffer is dependent on the output device. The above examples assumed a disk file with 256-word buffers. Other peripherals may use different sizes for their buffers.

APPENDIX F

STANDARD HARDWARE BOOTING PROCEDURES FOR TEK SPS BASIC VØ2

M9301 Bootstrap ROM Card

A. Perform either 1 or 2 below.

1. On a controller without switch registers:

   Press CNTRL-BOOT.

2. On a controller with switch registers:

   a. Press HALT.
   b. Enter the bootstrap address (usually either 173ØØØ or 173Ø1Ø) on the switch registers.
   c. Press LOAD ADDRESS.
   d. Press ENABLE.
   e. Press START.

B. In response to the prompt character ($) printed on the terminal, type the device name and drive number, followed by a carriage return. The devices supported by the M9301 and TEK SPS BASIC VØ2 are:

DXn	where n is a Ø or 1
DKn	where n is an integer between Ø and 7, inclusive.

M9312 Bootstrap ROM Card

A. Perform either 1 or 2 below.

1. On a controller without switch registers:

   Press CNTRL-BOOT.

2. On a controller with switch registers:

   a. Press HALT.
   b. Enter the bootstrap address (usually either 173ØØØ or 173Ø1Ø) on the switch registers.
   c. Press LOAD ADDRESS.
   d. Press ENABLE.
   e. Press START.

B. In response to the prompt character ($) printed on the terminal, type the device name and drive number, followed by a carriage return. The devices supported by the M9312 (with the required ROM) and TEK SPS BASIC VØ2 are:

DXn	where n is a Ø or 1
DKn	where n is an integer between Ø and 7, inclusive.
DLn	where n is an integer between Ø and 3,inclusive (not supported by TEK SPS BASIC VØ2-Ø1)
DYn	where n is a Ø or 1 (not supported by TEK SPS BASIC VØ2-Ø1)

Standard ROM Bootstrap on SBT Module in CP4165

A. Press RESTART.

B. In response to the prompt (DEV=) printed on the terminal, type:

DXn where n is a Ø or 1

Do not enter a carriage return.

APPENDIX G

A METHOD FOR MORE ACCURATE TIMING WITH WAIT AND SIFTO

Due to the different types of memory available (including cache), the timing in the WAIT and SIFTO commands is not always as accurate as some applications demand. Here are some simple methods to calculate a calibration ratio specific to your system by using a stopwatch. The ratio can be used to adjust the WAIT and SIFTO commands for more accurate timing.

To calculate a ratio for the WAIT command, use the following program:

```
10 LOAD 'WAIT','PRINT'
20 PRINT 'WHEN YOU ARE READY, SIMULTANEOUSLY HIT A KEY'
30 PRINT 'AND TURN ON THE STOPWATCH'
40 PRINT 'WHEN THE TERMINAL BELL RINGS, TURN OFF THE STOPWATCH'
50 WAIT
60 WAIT 30000
70 PRINT '^G'
80 PRINT 'INPUT TIME FROM THE STOPWATCH'
90 INPUT T
100 PRINT 'THE WAIT-ADJUSTMENT RATIO= ';30/T
```

The timing portion of the program takes 2Ø-4Ø seconds to complete; exact time depends upon the computer type. Now, whenever you use the WAIT command with this system, multiply the time you want to wait by the WAIT Adjustment Ratio.

To calculate a ratio for the SIFTO command, use the following program. A GPIB instrument should be connected to the interface and turned on. Lines 2Ø and 8Ø should use the interface number your hardware is strapped for.

```
10 LOAD 'GPI','RASCII','SIFTO','PRINT','ONERR','WAIT'
20 SIFTO @0,30000
30 ONERR Z GOTO 100
40 PRINT 'WHEN YOU ARE READY, SIMULTANEOUSLY HIT A KEY'
50 PRINT 'AND TURN ON THE STOPWATCH'
60 PRINT 'WHEN THE TERMINAL BELL RINGS, TURN OFF THE STOPWATCH'
70 WAIT
80 RASCII A$ FROM @0
90 RETURN
```

```
100 PRINT '^G'
110 PRINT 'INPUT TIME FROM THE STOPWATCH'
120 INPUT T
130 PRINT 'THE SIFTO ADJUSTMENT RATIO= ';30/T
```

The timing portion of the program takes 20-40 seconds to complete; exact time depends upon the computer type. Now, whenever you use the SIFTO command, multiply the timeout value you want to specify by the SIFTO Adjustment Ratio.

A convenient way to implement this in a program is to set a variable to the value of the calculated adjustment ratio. Then each time the SIFTO command is used in the program, specify the time-out value with the desired time-out period multiplied by that variable. If the program is run on another computer, only the single line defining the variable needs to be changed.

APPENDIX H

SIZES OF TEK SPS BASIC VØ2/VØXM NONRESIDENT COMMANDS AND DRIVERS

The memory size listed for each nonresident command and driver refers to the approximate number of words of controller memory required to load that particular module. This size may change with a new release of the module. The amount of memory needed to execute the command or driver may be considerably more.

Approximate Size of TEK SPS BASIC VØ2 Modules

VØ2 Module	Size in Words	VØ2 Module	Size in Words
ABORT.SPS	28	DK.SPS	1551
ADLOG.SPS	657	DL.SPS	1714
ADPLOT.SPS	1809	DLOG.SPS	984
ATAN2.SPS	371	DPO.SPS	3911
BITCLR.SPS	126	DRAW.SPS	67
BITSET.SPS	126	DRAWON.SPS	14
BITTST.SPS	146	DX.SPS	1545
BOOT.SPS	244	DY.SPS	1620
CANCEL.SPS	221	EDGE.SPS	484
CHAIN.SPS	144	EDGEAD.SPS	378
CHANGE.SPS	908	ENVDPO.SPS	317
CLEAR.SPS	126	EOF.SPS	37
CLK.SPS	392	FORMAT.SPS	275
CONVL.SPS	1579	GET.SPS	99
COPY.SPS	584	GETBLK.SPS	142
CORR.SPS	1667	GETFRE.SPS	48
CT.SPS	1091	GETLIN.SPS	25
DATE.SPS	199	GETLOC.SPS	206
DAVG.SPS	957	GETPRI.SPS	26
DEFECT.SPS	177	GETR5.SPS	37
DEFINE.SPS	185	GETSTA.SPS	243
DEVCLE.SPS	170	GIFES.SPS	62
DIFF.SPS	439	GIN.SPS	155
DIR.SPS	653	GPI.SPS	1051
DISPLA.SPS	355	GRAPH.SPS	2209

VØ2 Module	**Size in Words**	**VØ2 Module**	**Size in Words**
HASH.SPS	126	ONERR.SPS	165
HINPUT.SPS	279	OPRINT.SPS	399
HOOK.SPS	1070	OSET.SPS	123
HOOKQ.SPS	1103	OVERLA.SPS	111
HPRINT.SPS	407	OVLOAD.SPS	499
HSET.SPS	133	OVLSAV.SPS	415
IFDTM.SPS	121	PAGE.SPS	26
IGNORE.SPS	97	POLAR.SPS	587
INITG.SPS	28	POLL.SPS	334
INPREQ.SPS	114	PP.SPS	459
INPUT.SPS	313	PPOLL.SPS	64
INS.SPS	3008	PR.SPS	205
INSTAD.SPS	663	PRINT.SPS	307
INSTAL.SPS	663	PRIORI.SPS	14
INT.SPS	234	PUT.SPS	72
IV.SPS	313	PUTBLK.SPS	151
KBE.SPS	1146	PUTLOC.SPS	126
KBG.SPS	1099	RANDOM.SPS	71
KBN.SPS	855	RASCII.SPS	340
KBT.SPS	852	RBYTE.SPS	71
LASTST.SPS	103	RDRAW.SPS	67
LIST.SPS	609	READBI.SPS	448
LISTVA.SPS	573	READU.SPS	347
LOCAL.SPS	170	REJECT.SPS	115
LOCKKB.SPS	19	RENAME.SPS	239
LOCKOU.SPS	56	RENUM.SPS	284
LOCKSR.SPS	23	REPLAC.SPS	158
LP.SPS	340	RESCHE.SPS	94
LST.SPS	828	RESET.SPS	46
MAP.SPS	1396	RESETG.SPS	28
MAPAD.SPS	1396	REWIND.SPS	22
MATCH.SPS	173	RFFT.SPS	1643
MODE.SPS	119	RFFT1.SPS	1392
MOVE.SPS	65	RMOVE.SPS	65
MT.SPS	1206	RSDRAW.SPS	67
NORMAD.SPS	610	RSMOVE.SPS	65
NORMAL.SPS	535	RSTBUS.SPS	65
ODT.SPS	1738	SAVE.SPS	146
OINPUT.SPS	269	SCHEDU.SPS	361

VØ2 Module	Size in Words	VØ2 Module	Size in Words
SDRAW.SPS	67	TIME.SPS	241
SEEVIE.SPS	95	TRIGGE.SPS	170
SEEWIN.SPS	95	UNLOG.SPS	645
SETDAT.SPS	185	UNSCHE.SPS	71
SETGR.SPS	433	VARADR.SPS	53
SETTIM.SPS	217	VARCLR.SPS	188
SGIN.SPS	143	VARSET.SPS	188
SIFCOM.SPS	184	VARTST.SPS	147
SIFLIN.SPS	192	VERSIO.SPS	268
SIFTO.SPS	39	VIEWPO.SPS	89
SMOVE.SPS	65	VM.SPS	1333
SQUISH.SPS	1800	WAIT.SPS	81
SRQDIS.SPS	64	WASCII.SPS	333
SRQENA.SPS	71	WBYTE.SPS	106
STAT.SPS	798	WHEN.SPS	101
STATUS.SPS	745	WINDOW.SPS	78
STERMC.SPS	68	WRITE.SPS	143
SYSBLD.SPS	867	WRITEU.SPS	394
TD.SPS	2353	XYPLOT.SPS	2094
TDPLOT.SPS	1817	ZERO.SPS	55
TIFL.SPS	59	ZREF.SPS	207

Approximate Size of TEK SPS BASIC VØ2XM Modules

VØ2XM Module	Size in Words	VØ2XM Module	Size in Words
ABORT.SPS	28	GET.SPS	99
ADLOG.SPS	657	GETBLK.SPS	152
ADPLOT.SPS	1825	GETFRE.SPS	76
ATAN2.SPS	411	GETLIN.SPS	30
BITCLR.SPS	145	GETLOC.SPS	230
BITSET.SPS	145	GETPRI.SPS	31
BITTST.SPS	169	GETR5.SPS	38
BOOT.SPS	244	GETSTA.SPS	257
CANCEL.SPS	221	GIFES.SPS	73
CHAIN.SPS	144	GIN.SPS	161
CHANGE.SPS	908	GPI.SPS	1053
CLEAR.SPS	126	GRAPH.SPS	2209
CLK.SPS	392	HASH.SPS	126
CONVL.SPS	1996	HINPUT.SPS	288
COPY.SPS	584	HOOK.SPS	1070
CORR.SPS	2154	HOOKQ.SPS	1103
CT.SPS	1091	HPRINT.SPS	407
DATE.SPS	212	HSET.SPS	140
DAVG.SPS	1040	IFDTM.SPS	121
DEFECT.SPS	180	IGNORE.SPS	97
DEFINE.SPS	185	INITG.SPS	28
DEVCLE.SPS	170	INPREQ.SPS	114
DIFF.SPS	515	INPUT.SPS	329
DIR.SPS	653	INS.SPS	3165
DISPLA.SPS	355	INSTAD.SPS	725
DK.SPS	1551	INSTAL.SPS	725
DL.SPS	1714	INT.SPS	283
DLOG.SPS	984	IV.SPS	313
DPO.SPS	3990	KBE.SPS	1146
DRAW.SPS	67	KBG.SPS	1099
DRAWON.SPS	14	KBN.SPS	855
DX.SPS	1545	KBT.SPS	852
DY.SPS	1620	LASTST.SPS	116
EDGE.SPS	602	LIST.SPS	609
EDGEAD.SPS	489	LISTVA.SPS	573
ENVDPO.SPS	317	LOCAL.SPS	170
EOF.SPS	37	LOCKKB.SPS	19
FORMAT.SPS	275	LOCKOU.SPS	56

VØ2XM Module	**Size in Words**	**VØ2XM Module**	**Size in Words**
LOCKSR.SPS	23	RENUM.SPS	284
LP.SPS	340	REPLAC.SPS	158
LST.SPS	828	RESCHE.SPS	94
MAP.SPS	1549	RESET.SPS	46
MAPAD.SPS	1549	RESETG.SPS	28
MATCH.SPS	178	REWIND.SPS	22
MODE.SPS	119	RFFT.SPS	2014
MOVE.SPS	65	RFFT1.SPS	1677
MT.SPS	1206	RMOVE.SPS	65
NORMAD.SPS	665	RSDRAW.SPS	67
NORMAL.SPS	571	RSMOVE.SPS	65
ODT.SPS	2231	RSTBUS.SPS	65
OINPUT.SPS	278	SAVE.SPS	146
ONERR.SPS	165	SCHEDU.SPS	361
OPRINT.SPS	399	SDRAW.SPS	67
OSET.SPS	130	SEEVIE.SPS	108
OVERLA.SPS	111	SEEWIN.SPS	108
OVLOAD.SPS	500	SETDAT.SPS	185
OVLSAV.SPS	415	SETGR.SPS	433
PAGE.SPS	26	SETTIM.SPS	217
POLAR.SPS	670	SGIN.SPS	149
POLL.SPS	338	SIFCOM.SPS	184
PP.SPS	459	SIFLIN.SPS	192
PPOLL.SPS	71	SIFTO.SPS	39
PR.SPS	205	SMOVE.SPS	65
PRINT.SPS	307	SQUISH.SPS	1800
PRIORI.SPS	14	SRQDIS.SPS	64
PUT.SPS	72	SRQENA.SPS	71
PUTBLK.SPS	151	STAT.SPS	858
PUTLOC.SPS	140	STATUS.SPS	806
RANDOM.SPS	76	STERMC.SPS	68
RASCII.SPS	345	SYSBLD.SPS	960
RBYTE.SPS	76	TD.SPS	2415
RDRAW.SPS	67	TDPLOT.SPS	1827
READBI.SPS	554	TIFL.SPS	66
READU.SPS	362	TIME.SPS	254
REJECT.SPS	153	TRIGGE.SPS	170
RENAME.SPS	239	UNLOG.SPS	707

VØ2XM Module	**Size in Words**	**VØ2XM Module**	**Size in Words**
UNSCHE.SPS	71	WASCII.SPS	333
VARADR.SPS	79	WBYTE.SPS	106
VARCLR.SPS	202	WHEN.SPS	101
VARSET.SPS	202	WINDOW.SPS	78
VARTST.SPS	158	WRITE.SPS	143
VERSIO.SPS	268	WRITEU.SPS	394
VIEWPO.SPS	89	XYPLOT.SPS	2112
VM.SPS	1260	ZERO.SPS	55
WAIT.SPS	81	ZREF.SPS	153

YOUR COMMENTS COUNT

The Manual Writers at Tektronix, Inc. are interested in what you think about this manual, how you use it, and changes you might like to see in future manuals. Any queries regarding this manual will be answered personally.

What did you find that was:

interesting? __________

frustrating? __________

helpful? __________

confusing? __________

Is there anything you would like to see added to or deleted from this manual? __________

What is your major application area for this product? __________

Have you found any interesting applications, operating hints, or software routines which you would like to share with us? __________

*   *   *   *   *

Name:	Position:
Company:	Department:
Street:	
City:	State: Zip:

Fold on dotted lines and tape.
Postage will be paid by Tektronix, Inc. if mailed in U.S.A.

FIRST CLASS
PERMIT NO. 61
BEAVERTON, OREGON

BUSINESS REPLY MAIL

No postage necessary if mailed in the United States

Postage will be paid by

TEKTRONIX, INC.
P.O. Box 500
Beaverton, Oregon 97005

ATTN: SPS Documentation Group 157 — 94-384

Tektronix
COMMITTED TO EXCELLENCE

MANUAL CHANGE INFORMATION

Date: 6-10-80 Change Reference: C1/680

Product: TEK SPS BASIC VØ2/VØ2XM CP57000/CP57500 Manual Part No.: 070-2501-00

DESCRIPTION

MANUAL CHANGE INFORMATION

REV A, JUNE 1980

p. vii Page number corrections in **SECTION** 6:

Change "A typical System	6-1" to read 6-2
Change "Data Lines	6-4" to read 6-5

p. 1-8 New paragraph--insert above 1st paragraph beginning "Figure 1-2 shows a..."
[NOTE: This addition moves text and necessitates new page masters for p. 1-10, 1-11, and 1-12.]

The system software includes peripheral overlay files which also have the .OVL file-name extension. These .OVL files are used by certain peripheral commands, such as HOOK and HOOKQ, to supplement the device-driver modules. Unless the specified device is DX or DK, in order for such commands to execute properly, the corresponding .OVL file (e.g., DL.OVL and DY.OVL) must be on the system device or the target device. If you make your working copy of the system software by following the archiving procedures outlined in Appendix B, the proper files will be stored on your working disk.

p. 3-2 New paragraph--insert above **"NOTE"**

In order for LOADER to load .LDA files other than SPSDX.LDA and SPSDK.LDA, the appropriate peripheral overlay file for the device (e.g., DL.OVL and DY.OVL) must be on either the current system device or the target device.

p. 4-47 New paragraph--insert above **"Using the Syntax Options:"**
[NOTE: This addition moves text and necessitates new page master for p. 4-48]

If the first line of a scheduled routine is DELETEd, the entry in the Scheduler is updated to point to the line following the DELETEd line(s). (The function and parts of the Scheduler are explained in Section 1.)

Product: TEK SPS BASIC VØ2/VØ2XM Date: 6-10-80 Change Reference: C1/680

DESCRIPTION

p. 4-242 Text addition changes paragraph following **"Using the Syntax Option:"** to read:

The optional expression specifies the approximate number of milliseconds (1/1000 of a second) that program execution WAITs. The expression, when evaluated, is rounded to an integer. The largest acceptable result is 32767. If the expression is omitted, the resulting untimed pause must be terminated by a keyboard interrupt.

p. 4-254 Text correction replaces 2nd paragraph beginning "[WRITEU does not..."

[WRITEU does not output the TEK SPS BASIC data descriptors as the WRITE command does. Nor does WRITEU output any delimiters between data items, such as a carriage return, the way the PRINT command does. For this reason, the files output by WRITEU are sometimes called unformatted binary files.]

p. 6-1 Text addition--insert above the title **"Introduction to the IEEE 488 Bus"**:
[NOTE: This addition moves text and necessitates new page masters for p. 6-1 through 6-5.]

NOTE

Before attempting to control an instrument via this driver and its commands, consult the GPIB programming information in the manual for the instrument.

p. 6-67 Text addition changes paragraph following **"Using the Command Syntax:"** to read:

The expression following the at sign (@) is the number of the IEEE 488 interface which is assigned the time-out value. The second expression specifies the time-out value in milliseconds. The driver default time is 5 milliseconds. The largest acceptable time-out value is 32767 milliseconds. Specifying -1 indicates that the driver should wait indefinitely.

Product: TEK SPS BASIC VØ2/VØ2XM Date: 6-10-80 Change Reference: C1/680

DESCRIPTION

p. G-1 Text addition to end of 3rd paragraph beginning "The timing portion..."

(The longest time you can specify in a WAIT command is 32767 milliseconds. If the product of the WAIT Adjustment Ratio times the desired waiting time is greater than 32767 milliseconds, use more than one WAIT statement.)

p. G-2 Text addition to end of 1st paragraph beginning "The timing portion..."

(The longest time-out you can specify in a SIFTO command is 32767 milliseconds. If the product of the SIFTO Adjustment Ratio times the desired time-out value is greater than 32767, use more than one SIFTO statement.)

ATTENTION:

TEK SPS BASIC VØ1 Users

Converting to VØ2

A BASIC Conversion Program

There are some differences between TEK SPS BASIC VØ1 and TEK SPS BASIC VØ2. These differences should not affect most of your programs, but in case they do, we have included a BASIC program on your disk or diskette to aid you in converting your VØ1 programs to VØ2. You should find the conversion program, VØ1VØ2.CNV, easy to use. Just OLD in and RUN VØ1VØ2.CNV under VØ2 TEK SPS BASIC. Then, when the appropriate message(s) appears on the terminal, type the name of the program to be converted and the device on which it resides.

NOTE: Before running VØ1VØ2.CNV for the first time, make this simple change to it. Bring the conversion program into controller memory by typing:

```
OLD "VØ1VØ2.CNV""
```

Then add the line

```
16Ø35 IF LEN(IL$) = Ø THEN GOTO 16Ø7Ø
```

and save the updated version:

```
REPLACE "VØ1VØ2.CNV"
```

If this change is not made and the program being converted does **not** use the IEEE 488 Interface driver (GPI.SPS), the conversion program will change **all** DETACH statements to REM statements.

As VØ1VØ2.CNV changes your program, it will log all the changes to the terminal and/or line printer. If VØ1VØ2.CNV cannot change a line of the program, a warning message is printed to identify the line.

When the changes are completed, you may request that a new program file be created, with all changes included. The file may be written to any file-structured device supported by SPS.

If your program uses the IEEE 488 Interface driver (GPI.SPS) and commands, the VØ1 syntax can be converted to VØ2 syntax after you supply the IEEE 488 Interface number and the ILUN used in VØ1 to attach the interface.

What the Conversion Program Can Handle

The following is a list of exceptions to compatibility between VØ1 and VØ2 user BASIC programs that the conversion program can detect. The way the conversion program handles these exceptions is detailed within square brackets.

1. For VØ2, the cassette is no longer supported as a system device. SPSCT.LDA is not sold with the main package. The BOOT command will not work with a cassette. [If a program being converted has a BOOT CT: statement, a message is printed and the statement is changed to a remark (REM statement).]

2. The VØ2 ONERR command has been improved to take usage into account, so an ONERR statement may not function exactly as it would under VØ1. [The conversion program flags all ONERR statements by printing the statement on the logging device with a message to check the usage.]

3. The syntax of the VØ2 SCHEDULE and UNSCHEDULE commands have been changed to allow the proper renumbering of the referenced line numbers with the RENUM command. [In all cases but one, the conversion program correctly changes the syntax. If a line number is specified by an expression, however, the syntax is changed, but a message is printed so that you can change the expression to an actual line number.]

4. For VØ2, the optional commas in the OLD, LIST, and SAVE commands were made mandatory. [The conversion program is able to change the syntax in some cases, and prints a message in all others.]

5. The scale factor is processed differently by the VØ2 TDPLOT than by the VØ1 TDPLOT. This change was necessary for consistency with ADPLOT [The conversion program handles the difference.]

6. The VØ2 IEEE 488 Interface driver (GPI.SPS) is different. There are two major changes in GPI.SPS:

a) The driver is not ATTACHed or DETACHed in VØ2. [The conversion program can delete all such occurrences from VØ1 programs.]

b) The GET, PUT, WHEN and IGNORE commands and the nonresident commands which call GPI.SPS do not use the pound sign (#) to talk to the driver. [The conversion program changes the pound sign to an at sign (@) and inserts the correct interface number.]

What the Conversion Program Can't Handle

There are two general areas where programs written for VØ1 may not run correctly under VØ2. These cannot be corrected by the conversion program:

1. Timing.

2. Avoidance of error messages or known bugs by "programming around" them. Some common problem areas are:

a) The computed GOSUB and computed GOTO commands have been changed so that if the line number selector is less than 1 or greater than the number of line numbers in the list of line numbers, the GOSUB or GOTO is ignored. No warning is issued. (VØ1 software issued a fatal error if the line number selector was out of range.)

b) The waveform units processing is simpler in VØ2 than in VØ1. The main difference is in error handling. In the following cases, assume that W1, W2 are waveforms, A is an array, and V is a scalar:

1) In the case of A/W1, VØ1 produced vertical units the same as W1's. VØ2 produces vertical units the inverse of W1's. There is no impact to your program unless the vertical units produced by VØ1 bothered you and you implemented changes in your programs. In this case, running the programs under VØ2 will produce errors in vertical units.

2) In any of the following cases, VØ1 output a delta (Δ) to indicate potential incompatibilities between units,

while VØ2 does not. There is no impact to your program unless you deliberately changed the units to accommodate the delta:

W1*A	A+W1	W1↑A	W1/V
A*W1	W1-A	A↑W1	W1+V
W1/A	A-W1	V↑W1	W1-V
A/W1	W1↑V	W1*V	V+W1
W1+A	W1↑W2	V*W1	V-W1