Application Note 417-1

HP BASIC I/O Concepts

For the HP 9000 Series 200 and Series 300

HEWLETT PACKARD

Preface

This document introduces the I/O features implemented in HP BASIC which make the language a very powerful tool for the successful development and completion of an I/O process. It is intended for individuals who are investigating the use of a controller to automate their measurement, data acquisition or test process and who are interested in the tools a controller language can offer.

Table of Contents

HP BASIC I/O Introduction

When looking for an instrument controller it is easy to overlook a system's language capabilities. Hardware configuration and pricing are, of course, very important, but the most valuable assets an instrument controller can offer are the language capabilities and program development tools it provides. These can yield the highest returns or turn an investment into a painful, costly obstacle to the successful completion of any application.The use of a language that does not provide a thorough enough combination of simple, flexible and high performance I/O and programming tools will result in increased development time, often in reduced performance and even in the inability to complete or carry out an I/O process.

An I/O process usually consists of setting up a resource or device, making a measurement and then either storing and presenting the data or making adjustments to or decisions about a device and repeating the measurement. The roles of an I/O friendly language are many-fold:

- To provide adequate performance and be able to interface with very slow or very fast devices without loss of speed or data
- To posess sophisticated enough drivers to communicate with different interfaces transparently in addition to providing the user with the information and tools to control the process from the program level
- To provide tools to customize an I/O process by means of different ways of formatting data, specifying the amount and size of the transfer and the way a transfer is to terminate
- To provide high-level commands to easily process interrupts or specify events of significance to the program
- To offer a vast array of programming and editing tools to easily develop programs, make changes and allow for re-useable sections of code
- To provide computational and graphics tools to process and present the measurement in a significant manner.

The purpose of this document is to explain the features unique to HP BASIC, Hewlett-Packard's own enhanced implementation of Technical BASIC for the Series 200 and 300 computers, which make the language a simple yet powerful tool to completely and succesfully carry out the I/O process described. These features, their implementation and programming will be described in the remainder of this paper. Furthermore, some of the computational, editing, development and graphics tools will be summarized and an appendix containing a list of supported interfaces will be included.

The I/O tools explained in the following sections are:

Unified I/O – a language implementation that allows for the same statements to be used with different types of devices, thus providing the capability to easily redirect I/O and change devices without major code changes.

I/O Paths – a method of assigning a variable name to a device in order to customize data transfers, easily change devices with only a one-statement change to a program or simply to improve performance by up to 30%.

Data Formatting – a method provided to allow the user to specify exactly how the data should look as it is entered into the controller, sent to an instrument or output to the screen, a printer, a plotter or a file.

Memory Mapped I/O and Registers – an implementation that allows for easy access to driver or hardware information and control mechanisms without the complexity of Assembly language programming.

Transfers and Buffering – enhanced methods to input and output data which not only solve the problems of interfacing to very slow devices or losing data from very fast devices but provide improved performance by overlapped (background) execution.

Event and Interrupt Programming – a group of very simple statements and tools which allow for even the most sophisticated applications to interactively control and interface to devices without the painful process of Assembly language programming.

Unified I/O

The concept of unified I/O refers to a language implementation that provides the same statements for use with all resources. I/O in HP BASIC can be as easy as using the words ENTER and OUTPUT. These two I/O commands can be used to input and output data to just about any device or system resource connected to the Series 200 or 300 workstations. And most other I/O commands available through HP BASIC have the same characteristic: they can be used with several different types of devices and therefore there is no need to change or learn new commands. For example, the PRINTER IS statement allows the user to specify a default printer. This simple declaration allows for the target device to be changed very simply, without changing the physical code intended to deal with printers. It makes for such flexible coding that the same PRINT statement which first directed data to the screen may the next time go to an external printer for hardcopy or to a file for permanent storage. This same flexibility applies to many other HP BASIC commands.

As another example, the ENTER and OUTPUT statements are flexible enough to be used with several very different types of devices. These could include your computer's display, keyboard or memory, mass storage devices and instruments or peripherals connected to the computer via an interface card.

The syntax of the OUTPUT statement is as follows:

```
OUTPUT device_select_code; data1,data2,...
```

(The syntax of the ENTER statement is equivalent.)

This one-line statement alone will move data between the controller and a resource. The data can be in the form of numbers, characters, or arrays, either entered directly in the statement or through variables. The device select code is a number that uniquely identifies a device. Please refer to Figure 1 for examples of unified I/O as implemented in HP BASIC.

```
OUTPUT 1; "Hello, my name is ";Name$
```

The device at select code 1 is defined to be the screen. After this statement has been executed, the message, "Hello, my name is" will appear on the screen, followed immediately by the string contained in the variable. If Name$ contained the string "John Smith" then the following would appear on the screen:

```
"Hello, my name is John Smith"
```

```
OUTPUT 9; "This is a line of text",8,Data(*)
```

The string "This is a line of text" followed by the number 8 and the numbers contained in the array Data will be sent to the interface set to select code 9 – usually a serial (simple RS-232C) interface card. *(**Note:** the numbers will be sent in their ASCII representations. There are simple ways, outlined in later sections, to send numbers in internal representation. Most devices, however, prefer ASCII communication.)*

Figure 1 Examples of Unified I/O as Implemented by HP BASIC

From Figure 1, it can be seen that the command doesn't change when communicating with different devices, only the select code that specifies which device one wants to talk to.

Unified I/O makes the programming task and output redirection to another resource very simple and flexible. A user does not need to learn a vast array of commands to be able to communicate with different devices, unless the more sophisticated user wanted to access lower-level functions – tools are available for this as well. For example, there are over twenty dedicated commands for talking to devices on an HP-IB 8-bit parallel (IEEE-488) bus.

I/O Paths

All data moved into and out of the computer goes through an "I/O path". For example, when outputing data to a device, the path taken will be as follows: the operating system determines which command is to be executed and to what device, the system then looks for device information and attributes to communicate with the device, and finally, data is sent through the interface card hardware. Every time an I/O statement is used with an explicit number as the select code, the system will go through this path to finally talk to the device. Please refer to Figure 2 for a diagram of this default I/O path.

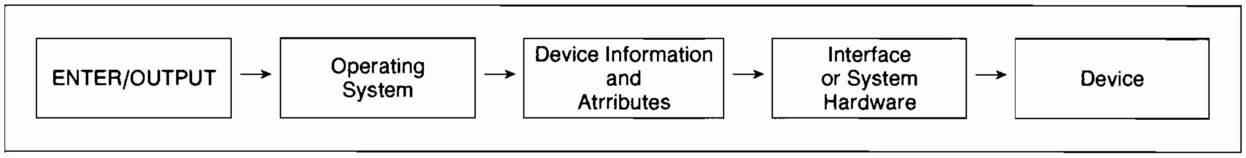
ENTER/OUTPUT
Operating System
Device Information and Atrributes
Interface or System Hardware
Device

Figure 2 Default I/O Path

HP BASIC provides a powerful tool which allows a user to assign a name to an I/O path, so as to prevent the system from going through the whole process every time an I/O operation is to be done. The I/O path name is similar to a variable name as all information about the variable and its access are stored in one place. Device and path information can thus be stored under a variable name and be accessed easily by the system every time the name is invoked. Please refer to Figure 3 for a diagram of the steps taken by HP BASIC when executing an I/O command if a path name has been assigned.

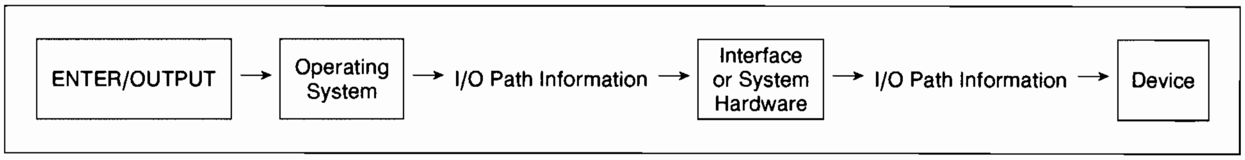

Figure 3 Defined I/O Path

There are many advantages to naming an I/O path. Because all the information about an I/O path is stored in and associated with the path name, the system no longer needs to create this information every time an I/O statement is executed. This leads to greatly increased performance. For I/O operations, this increase can amount to 30%. For instrument I/O, this increase will also be dependent on the speed of the instrument. In addition, more readable code results from using path names. In effect, since a path name can be thought of as a variable name that contains device information, if a device change needs to be made, only one statement in the whole program needs to be altered – the statement assigning the path name. The syntax of this statement is illustrated in Figure 4.

The syntax to assign an I/O path name is as follows:

```
ASSIGN @pathname TO select_code
```

Where the pathname can be any variable name and the select code can be any legal device locator

```
ASSIGN @Serialpath TO 9
```

The device at select code 9, by default an RS-232C serial card, has been assigned the name "serialpath". For any subsequent operations, the select code may be replaced by the path name. For example:

```
OUTPUT @Serialpath; "hello",data(*)
```

If for example, a Datacomm card at select code 20 were to replace the serial card at select code 9, the only change to the program would be in the ASSIGN statement:

```
ASSIGN @Serialpath TO 20
OUTPUT @Serialpath; "hello",data(*)
```

The OUTPUT statement would remain unchanged.

Figure 4 Examples of Assigning an I/O Path Name in a Program

In addition to a significant performance gain and more readable code, an I/O path name provides the added flexibility of modifying default attributes. For example, you can specify whether you want your data to be handled and stored in ASCII or internal format. In actuality, all the data is handled in internal format, but it will go through an ASCII formatting routine as a default for many devices. This process, of course, takes more time and in some applications, may not be necessary. Another important attribute that can be changed is the number of bytes transferred per cycle. If you are using a 16-bit interface, you may want transfers to occur in 16-bit (word-wide) cycles to improve performance. Or, you may have a device that can only send or receive data in byte-wide increments and you want to use only eight data lines from your interface bus. Other attributes you can specify include the use of a conversion table for characters you may want to change as they are entered, specifying or changing characters to end a transfer, or controlling parity generation and checking. Please refer to Figure 5 for an example of changing attributes with an ASSIGN statement.

```
ASSIGN @Serialpath TO 9; FORMAT OFF
```

This statement will disable the ASCII formatting routine and will cause the system to handle data being sent through "serialpath" in internal representation only. To specify ASCII format, you need only add FORMAT ON to the ASSIGN statement

```
ASSIGN @Gpiopath TO 12; WORD
```

The WORD parameter, and its equivalent, BYTE, allow the user to specify whether transfers will take place in word-wide or byte-wide increments. The GPIO bus is a 16-bit interface, and specifying the WORD parameter allows transfers to use all 16 bus lines

Figure 5 Setting Default Attributes Using the ASSIGN Statement

Data Formatting

Even though the parameters of FORMAT ON/OFF are useful, they may not be specific enough for an application. For example, a device may expect data in scientific notation, with a one digit exponent or with no end of line sequences. The user may want to store data coming from a device in a special format, mixing character strings with numbers. With an added parameter to the ENTER and OUTPUT statements, this can be done very easily. HP BASIC provides additions to its I/O statements and a very complete list of tools to format data so that transfers to devices, including storage files or the CRT, can be as customized as the user wants them to be.

The OUTPUT USING and the ENTER USING statements allow for the desired format of data to be output or entered. This can be done through image specifiers which either describe the desired format of each item in the data list or specify that special characters are to be output. The syntax is as follows:

```
OUTPUT select_code/path_name USING image_specifier_list
ENTER select_code/path_name USING image_specifier_list
```

Image specifiers are character codes that can be joined together thus forming an "image" of what the data should look like. Some of the codes available are:

D Specifies that one digit is to be output in ASCII form or that one byte is to be entered and interpreted as a numeric character also in ASCII form

E Specifies that the number is to be output in scientific notation

S Specifies that a leading "+" or "–" be output for sign of the number

A Specifies that one character is to be output or that a character is to be entered and interpreted as a string character

X Specifies that a space is to be output or that a character is to be entered and ignored

Specifies that a statement-termination condition is not required, the ENTER statement is terminated as soon as the last image item is satisfied. With an output statement, it specifies that the current end of line sequence which normally follows the last item is to be suppressed.

The preceding is by no means an exhaustive list, merely a few statements to better illustrate the concept of image specifiers. An example is shown in Figure 6.

```
OUTPUT @Crtpath USING "6A,SDDD.DDD,3X"; " k= ",123.4
```

This image specifies that the data should be output as six characters, a signed number with three digits in front of the decimal point and three following the decimal point, and three spaces after the number. For the first six characters, if the number of characters actually output is smaller than six, then trailing blanks will be added. The resulting output on the screen will look like this:

```
_k=___+123.400___
```

(The underline characters do not appear in the screen but are used here to denote blank spaces)

The image specifier list, aside from being directly placed in the OUTPUT USING or ENTER USING statements, can take the form of a pre-defined string variable, a line label or a line number where the definition appears:

```
String$="6A,SDDD.DDD,3X"
OUTPUT @Crtpath USING String$; " k= ",123.4
```

The previous example uses a string variable to specify the image. This method, along with the ones used in the following examples, has the advantage that it is re-useable. The string need only be declared once and all I/O statements can use it without the need of describing the image list over and over.

The following two methods use the IMAGE statement to specify that the particular program line is not a command but merely a description of the image specifier list. The line number or label can then be used to invoke the list from an I/O statement.

```
OUTPUT @Crtpath USING 100; " k= ",123.4
100 IMAGE 6A,SDDD.DDD,3X

OUTPUT @Crtpath USING Image_spec; " k= ",123.4
Image_spec: IMAGE 6A,SDDD.DDD,3X
```

Figure 6 Examples of Data Formatting

Memory Mapped I/O and Registers

The Series 200 and 300 computers employ "memory-mapped" I/O operations. This means that a device can be accessed just as if it were a memory location. Within these locations, there are registers associated with each device. Some registers store parameters that describe or control the operation of an interface, some store information describing the I/O path to a device, and some store the memory locations at which interface cards reside.

Most devices and I/O path names have their own set of registers where information pertinent to each is stored. For example, register 0 of a display holds the "x" screen coordinate where data is to be displayed; register 21 of the display allows you to choose between bit-mapped mode or separate alpha and graphics rasters, depending on your system's hardware. For a serial interface card, register 1 allows you to reset the card and register 3 allows you to set a new baud rate for information transfer. There are many registers that are uniquely associated with each device, interface card or I/O path name – all well documented and available to the user.

There are different ways to access device registers. The system does it automatically when an I/O statement is executed. For user access, there are two sets of commands that can be used: one set involves getting at registers through the software drivers and the other involves interrogating the hardware directly. It is easier to access registers through the drivers and the remainder of this discussion will deal with this method.

Register access through the drivers is done with two very simple commands – STATUS and CONTROL. STATUS is used to read registers that contain information about a device while CONTROL, as the name implies, is used to write to registers that have the ability to control or change the state of devices. The syntax for these statements and some programming examples are illustrated in Figure 7.

The syntax of the STATUS and CONTROL statements is as follows:

```
STATUS select_code/path_name, register_number; variable_name
CONTROL select_code/path_name, register_number; data
```

The register numbers are documented for each device and specify which register is to be accessed. Information registers return data through variable names included in the statement while most control registers expect a certain parameter that specifies what changes are to be made.

```
CONTROL 9,3; 9600
```

The device at select code 9 is usually the serial card. Register number 3 belonging to that card allows the user to change the data rate at which the transfer will take place. In the above example, the data or baud rate has been set to 9600 bits per second.

Figure 7 Using Device Registers

In many other languages, access such as this into interface card registers would require the tedious process of Assembly language programming. Clearly, the STATUS and CONTROL statements provide very easy and convenient access to most any information and control for a device or path name.

Advanced Transfer Techniques

While the ENTER and OUTPUT statements suffice for many applications, they may not perform adequately when dealing with very slow or very fast devices. The ENTER and OUTPUT statements do not release the computer until their execution is completed. With very slow devices, the program has to adjust to their speed – causing the overall execution time to be very slow. Conversely, very fast devices can have an adverse effect on program execution and data integrity. A fast device may attempt to send data faster than the computer can accept it, thus corrupting or losing the data. To overcome both problems, an alternate method has been implemented in HP BASIC – the TRANSFER statement.

Before a transfer takes place, an area of memory is reserved to hold the data being transferred. This area of memory is called a buffer and its operation is analogous to setting up a fast internal device. It takes the form of real array, an integer array or a string scalar,and an I/O path name can be assigned to it. Once a buffer has been created and an I/O path name assigned, data can be transferred into or out of the buffer with a TRANSFER statement.

There are many advantages to using the TRANSFER statement. First of all, the statement has been implemented to support background or overlapped execution. This means that when an I/O process is being carried out through the TRANSFER statement the system will allow the CPU to execute other statements simultaneously. Background execution not only guarantees greater overall program performance, but it is especially advantageous when interfacing with slow devices that normally would have slowed down program execution. High speed devices can also be dealt with successfully. Since buffers are very fast internal devices, data entering the computer will not overrun the CPU and data integrity will be kept. Best of all, the TRANSFER statement does not have to be used with fast or slow devices exclusively. I/O speeds can be greatly improved when using buffers and the TRANSFER statement because data is moved by the fastest method available. In addition, it is the only statement, aside from disc I/O statements, that can automatically make full use of DMA (Direct Memory Access) card capabilities.

Figure 8 illustrates the flow of the transfer statement as it is executed and Figure 9 shows program lines exemplifying the minimum programming needed to set up and do a transfer from a device to a buffer and from a buffer to a file.

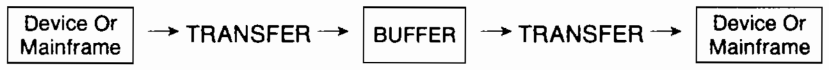

Figure 8 Flow of a TRANSFER Statement

```
DIMENSION Text$[256] BUFFER             ! set up buffer variable
ASSIGN @Buff TO BUFFER text$            !assign a path to the buffer
ASSIGN @Device TO 12                    !assign a path to the device
ASSIGN @File TO "Filename"              !assign a path to the file
TRANSFER @Device TO @buff               !transfer data from the device to the buffer
TRANSFER @Buff TO @file                 !transfer data from the buffer to a file
```

•

Other program statements

•

After assigning path names, a TRANSFER is set up between the device and a buffer and another between the buffer and a storage file. If at a particular moment no data is being transferred between the device and the buffer, the system will try to transfer data between the buffer and the file. If data is unavailable to both TRANSFERs, the system will attempt to execute valid program statements that follow.

Figure 9 Programming a TRANSFER

It should be noted that buffers can be used with ENTER and OUTPUT. Program execution, however, will not be overlapped and the performance not as high because of the difference in implementation of the I/O statements. The TRANSFER statement, as mentioned earlier, is the fastest method available for carrying out I/O processes.

As with ENTER and OUTPUT, there are parameters you can specify to customize the TRANSFER statement:

`CONT`	for a standard inbound transfer, data from the device is placed in a buffer and the transfer is deactivated when the buffer is full. The CONT parameter allows for a transfer to continue indefinitely. If a buffer is full, no more data is transferred to it until some data has been removed from it – in effect, it employs a circular buffer approach.
`WAIT`	defers program execution until a transfer is done. Allows for non-overlapped execution.
`COUNT`	allows you to specify how many bytes of information you want to transfer.
`DELIM`	allows a transfer to terminate after a specified character is received on inbound transfers.
`END`	allows a transfer to finish on a device's default terminator.

To better understand the power of the TRANSFER statement, Figure 10 shows some performance figures for an HP-IB 8-bit interface card using OUTPUT/ENTER, OUTPUT/ENTER with I/O paths and the TRANSFER statement.The figures are taken from a program that takes 4,000 readings from an HP 3437A system voltmeter and are measured in seconds. The program tested did not include any program lines following the TRANSFER statement. Therefore, no overlapped execution was possible. Had it been, the differences in time of execution would have been even more marked.

Time/4000 Readings (seconds)	
ENTER/OUTPUT	4.32
I/O paths	3.85
TRANSFER	1.45

Figure 10 Performance

Events and Interrupts

Often, one may want to pause normal program execution when a certain event occurs in order to process some related code. An event may be that a certain key is pressed, an error condition occurs or an interface card or device wants to interrupt and grab the attention of the CPU. With some languages, you may have to keep checking whether or not the event has occurred – this wastes time. With others, the only way you can determine that an event has occurred is to do it through an Assembly language routine. HP BASIC, however, has implemented some very simple statements that allow you to alert the CPU of the possibility of an event that might occur. Once alerted, the CPU will continue the normal execution of your program and as soon as the event occurs, the CPU will automatically stop executing and branch off to a specified section of your code that services the event. All this can be done with the use of one statement in your program: the ON EVENT statement. A subset of the ON EVENT statements follows:

`ON END`	when the end of a mass storage file is encountered.
`ON ERROR`	when a program execution error is sensed.
`ON KBD`	when a specified key is pressed.
`ON KEY`	when a currently defined softkey is pressed.
`ON KNOB`	when the knob is turned.
`ON TIMEOUT`	when the computer has not detected a handshake response from a device within a specified amount of time.
`ON INTR`	occurs when a specified (or any) interrupt is requested by a device or when an interrupt condition occurs at the interface.

Figure 11 illustrates the syntax of ON EVENT statements and includes an example of the ease of programming events or interrupts.

The syntax of the ON EVENT statement is as follows:

```
ON event   parameter, priority   Branch_To_Code
```

The event can be one of those specified above. Some events like the ON KEY, have a parameter, such as the number of the particular softkey that is to cause the event. The priority of the particular event in relation to other specified events is a number from 1 to 15 and is optional. In addition, the section of code to branch to must be specified. This branch can be a line number or label (using the GOTO statement), a subroutine (using a GOSUB statement), or a subprogram (using a CALL or RECOVER statement).

```
ON KEY 2,3 GOTO Service_routine
```

This statement will alert the CPU that when softkey number 2 is pressed it should branch to the line labelled service_routine provided no events with priority higher than 3 are being serviced at the moment.

Figure 11 Event Programming

If you want to "catch" a specific interrupt condition on an interface card, the ON INTR combined with the ENABLE INTR statement will perform that function. As mentioned earlier, most devices have registers associated with them. For most interfaces, there is an "interrupt-enable" register. By writing a specific bit pattern to the register, you can choose from a variety of signals to cause an interrupt.

Figure 12 illustrates an interrupt enable register for an HP-IB card and an example that sets up the program to expect an interrupt on the SRQ signal.

bit 15 Value = -32768		bit 3 Value = 8	bit 2 Value = 4	bit 1 Value = 2 SRQ	bit 0 Value = 1

If you wanted to interrupt on an SRQ signal (bit 1 at value = 2 of the interrupt enable register), the following two statements are all that are needed:

```
ENABLE INTR select_code; 2
ON INTR select_code CALL subprogram_name
```

The ENABLE INTR statement specifies the device and sets up the signals that should cause interrupts. It is possible to specify more than one signal to cause an interrupt on a particular card simply by adding the values of the bits that represent those signals. For example, using a value of 6 would enable both signals represented by bits 2 and 1 (value = 4 + 2) of the interrupt enable register. In addition there are simple statements available that allow you to determine which signal in particular has caused an interrupt.

Figure 12 Interrupt Programming

Sample Program

The following program illustrates both the TRANSFER and the ON EVENT statements when your system is interfaced to a voltmeter. The only statements that would change if you were interfacing to another instrument would be the set-up statements, lines 140 and 150 in the program, and the ASSIGN statement with the specific HP-IB address of the instrument. The program will interrupt on the SRQ signal of the HP-IB, do a data transfer to the computer, store the data in a file and plot the results.

```
10   DIM Volts(4000) BUFFER                  ** Dimension buffer array
20   !
30   ASSIGN @Dvm TO 724                      ** Assign a path to the voltmeter
40   ASSIGN @File TO "filename"              ** Assign a path to a file
50   ASSIGN @Buffer TO BUFFER volts          ** Assign a path to a buffer for
                                                TRANSFER
60   !
70   CLEAR @Dvm                              ** CLEAR command for voltmeter
80   Hpib=7                                  ** variable for HPIB address
90   Mask=2                                  ** Mask to enable interrupt on SRQ
100  !
110  ON INTR Hpib GOSUB Srq_service          ** Set up and enable HPIB interrupt on
                                                SRQ
120  ENABLE INTR Hpib;Mask
130  !
140  OUTPUT @Dvm;"D.4SN1SE7SR2T1F1"          ** Set up the voltmeter programmatically
150  LOCAL @Dvm
160  !
170  LOOP                                    ** Wait for event/interrupt to occur
180    DISP "waiting for interrupt"
190  END LOOP
200  STOP
210  !
220 Srq_service:  !                          ** Service the interrupt once it occurs
230  TRANSFER @Dvm TO @Buffer                ** transfer data from voltmeter to buffer
240  TRANSFER @Buffer TO @File               ** transfer data from buffer to file
250  WINDOW 1,4000,-1,1                      ** scale the screen and plot data
260  FOR I=1 TO 4000
270    PLOT I,Volts(I)
280  NEXT I
290  RETURN
300  END
```

Figure 13 Sample Program

Other Capabilities

The previous sections have been a summary of the main features HP BASIC has to offer to the I/O process. There is, however, more to a controller's language than its I/O capabilities. You may want to perform a series of computations on the data acquired from an instrument or you may want to display the data graphically. You may also be concerned with the ease of program development and the amount of modularity your code can achieve. Following is a brief summary of some of the many additional capabilities HP BASIC has to offer.

Editing and Program Development

The following is a list of features provided by HP BASIC to improve the ease of program development and improve program entry time.

- Full screen editing
- Syntax checking after each line entry
- Powerful editing commands such as the following:
  - **COPYLINES**. Reproduce lines of code that have already been written.
  - **MOVELINES**. Move lines of code to other positions within the program.
  - **INDENT**. The system automatically indents every level of nested conditions or loops throughout the program.
  - **CHANGE** and **CHANGE ALL**. Find line or string patterns within a program and replace them with another specified line or string.
  - **FIND** and **FIND ALL**. Find line or string patterns within a program.
- **Pre-run capability** that checks for many run-time errors before a program is actually executed, thus, saving significant debugging and developing time.

Modular Programming

- **GOTO** and **SUBROUTINE** constructs
- **SUBPROGRAM** constructs. Subprograms are in a separate context from the main program. They allow local variables, parameter passing, their own set of softkey definitions, data blocks and line labels. In addition, subprograms do not need to reside in the same file as the main program, they can be loaded from a different file. Thus, developing libraries of I/O or other routines to be used in many different programs is very simple using subprograms.

Graphics

- **Drawing Tools.**
  - Plot command to easily plot 2D or 3D graphs with integer or real numbers and arrays;
  - Scaling and clipping capabilities;
  - User-defined characters and scaling units;
  - Labeling, axes and tick mark specification commands.
- **Storage of Images.** Commands available to store and retrieve graphics images to and from files
- **Color Graphics.** HP BASIC's command set allows for a variety of operations with color graphics, including the capability to define a custom palette of colors
- **Hardware.**
  - Video cards and monitors with different resolution providing either monochrome or color graphics;
  - Support for both bit-mapped and separate alpha and graphics displays;
  - Support for the knob, mouse, graphics digitizing tablet and touchscreen;
  - Graphics dump capabilities to a large array of printers.

Computations

- integer and real arithmetic
- trigonometric functions
- complex math functions
- matrix and array operations
- user-defined functions
- binary and base arithmetic
- time and date functions

Appendix A

Interface Card	Part No.	Description
HP-IB	98624A	IEEE-488 standard used to interface to HP printers, plotters, mass storage devices and many instruments.
GPIO	98622A	16-bit parallel card used to interface to many non-HP devices.
BCD	98623A	Binary coded decimal card for instrument interfacing.
RGB Color Video	98627A	Allows you to connect an external graphics display to your system.
2-channel DMA	98620B	Direct memory access card. It off-loads the main processor by providing separate I/O channels with their own controller isolated from the CPU. It speeds information transfers.
High-Speed Disc Interface	98625A	A high-speed HP-IB card that greatly improves communication with high-speed discs.
Standard RS-232	98626A	Standard serial asynchronous communications interface.
Datacomm	98628A	A smart serial card with built-in buffer and microprocessor. Protocol management can be done through the card.
Low-cost RS-232	98644A	A serial card with fewer features and no hardware select switch options.
Programmable Datacomm	98691A	Used for installing customized protocols.
SRM	98629A	Shared Resource Manager. Allows local networking through shared mass-storage devices and peripherals.
Breadboard card	98630A	For sophisticated users who want to build their own interface. The card provides the appropriate mechanical and electrical compatibility with the system.
7-channel A to D	98640A	Analog to Digital card with seven input channels for data acquisition applications.
Floating Point Math Card	98635A	Implements floating point math calculation in hardware. Provides large speed increases.
EPROM Programmer	98253A	Eraseable programmable read-only memory system including a card that allows easy programming.

Conclusion

As a controller language, HP BASIC aids in the I/O process by implementing a comprehensive, understandable set of built-in I/O commands, graphics and computational statements and an array of user-friendly features to speed up and ease program development. It is the most enhanced version of BASIC available in the market today but by no means the most complex. In fact, as could be seen throughout this document, HP BASIC provides a rich set of simple one-line statements that perform many complex tasks. It has been optimized for the I/O process and for effectively interfacing with the more than 1400 HP instruments and peripherals plus non-HP devices connected to a Series 200/300 controller through a variety of interface cards.

To summarize, the following matrix lists the main I/O enhanced features HP BASIC implements and their benefits to the user.

Feature	Benefit
Unified I/O	■ Fast and easy output redirection ■ Ease in learning I/O commands ■ Flexibility in device and resource usage
I/O Paths	■ Greatly improved performance ■ Customized I/O transfers ■ More readable code
Data Formatting	■ Custom formatting of data in I/O transfers ■ Custom formatting for display of data ■ Flexibility to interface to many devices
Memory Mapped I/O	■ Easy access to interface driver and hardware information ■ Simple programming – no need to use Assembly Language ■ Simple methods implemented to control the I/O process
TRANSFER and Buffers	■ Greatly improved performance ■ Solves problems of data loss and maintains data integrity with fast devices ■ Improves program performance when interfacing to slow devices ■ Overlapped/background execution for greater speeds
Events and Interrupts	■ Simple commands available to process any events ■ No need for Assembly language programming ■ Allows the user up to 15 priority levels

HEWLETT PACKARD

Printed in U.S.A. 11/86
5954-8273