HP BASIC Advanced Programming Techniques

HEWLETT PACKARD

HP Part No. 98616-90204
Printed in USA

Printing History

First Edition—January 1994

Product Compatibility

This manual supports:

HP BASIC/WS (Version 6.2)
HP BASIC/UX 300/400 (Version 6.3)
HP BASIC/UX 700 (Version 7.0)
HP BASIC/DOS (Version 6.2)

Contents

Tables

1 Introduction

This manual provides selected HP BASIC programming techniques for the advanced programmer. For a general overview of HP BASIC programming, refer to the *HP BASIC Programming Guide*.

There are three implementations of HP BASIC, which are described below. The notations BASIC/WS, BASIC/UX, and BASIC/DOS are used throughout this manual to identify these implementations.

BASIC/WS	The Workstation implementation, which is a combined language and operating system that runs on HP 9000 Series 200/300 computers. (This is probably the most familiar implementation of HP Series 200/300 BASIC language.)
BASIC/UX 300/400 & 700	The HP-UX implementations of BASIC on HP9000 Series 300/400 and Series 700 computers, respectively. HP BASIC/UX is, essentially, the BASIC interpreter and part of the BASIC Workstation operating system that runs as a set of processes "on top of" the HP-UX operating system.
BASIC/DOS	The DOS implementation, which is essentially Workstation BASIC modified slightly to run on the HP Measurement Coprocessor. BASIC/DOS supports both the HP 82300 Measurement Coprocessor and the HP 82324 High-Performance Measurement Coprocessor. These coprocessors provide HP Series 200/300 computer architecture on a PC plug-in card.

Most of the programming techniques described in this manual are applicable to all implementations of HP BASIC. However, where there are specific differences, they will be identified.

2 More About Numeric Arrays

This chapter describes selected topics for the advanced programmer using numeric arrays. For general information about dimensioning and using numeric arrays, refer to chapter 4, "Numeric Arrays," in the *HP BASIC Programming Guide*.

Arrays and Arithmetic Operators

BASIC allows you to multiply, divide, add, and subtract scalars to an array, as well as to add, subtract, multiply, and divide one array to another. It is also possible for you to add all the elements in an array to produce a single result. This section covers a function and operations which allow you to perform these tasks with **INTEGER**, **REAL**, and **COMPLEX** data types.

Using the MAT Statement

All arithmetic functions involving arrays must be preceded by the MAT keyword. The specified operation is performed on each individual element in the operand array(s) and the results are placed in the result array. The result array must be dimensioned to be at least as large as the current size of the operand array(s). If it is of a different shape than the operand array(s), the system will redimension it. Given the array A below, note how these arithmetic functions are performed.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

To add 3 to each element of array A, you would use the following statement:

```
MAT B= A+(3)
```

The result of the above addition is array B below:

$$B = \begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

To divide each element of array B above by 2, you would use the following statement:

```
MAT C= B/(2)
```

The result of the above division is array C given below:

$$C = \begin{pmatrix} 2 & 2.5 & 3 \\ 3.5 & 4 & 4.5 \\ 5 & 5.5 & 6 \end{pmatrix}$$

To multiply each element in array C by a scalar expression, you would use a statement similar to the following:

```
MAT C= C*(1+1+1)
```

The above statement multiplied each element in array C by 3 and placed that result in array C as shown below:

$$C = \begin{pmatrix} 6 & 7.5 & 9 \\ 10.5 & 12 & 13.5 \\ 15 & 16.5 & 18 \end{pmatrix}$$

Note that the result array can be the same as the operand array. Also, the scalar must be enclosed in parentheses.

In addition to performing arithmetic operations with scalars, you can also add, subtract, divide and multiply two arrays together. Except for multiplication with an asterisk, which is described later, these functions proceed as follows: Corresponding elements of each operand array are processed according to the specified operation, and the result is placed in the result array. The two operand arrays must be exactly the same size though their particular subscript ranges can be different. For multiplication, use a period rather than an asterisk. Using arrays A and B above, the statement,

```
MAT D= A+B
```

would give the array:

$$D = \begin{pmatrix} 5 & 7 & 9 \\ 11 & 13 & 15 \\ 17 & 19 & 21 \end{pmatrix}$$

The statement,

```
MAT B= A.B
```

would give:

$$B = \begin{pmatrix} 4 & 10 & 18 \\ 28 & 40 & 54 \\ 70 & 88 & 108 \end{pmatrix}$$

Again, the dimensioned size of the result array must be as large as the current size of each operand array. The two operand arrays must be identical in shape and size, but not necessarily in subscript ranges. For instance, `A` and `B` could have been dimensioned:

```
10   DIM A(1:3,2:4),B(-1:1,0:2)
```

Performing Arithmetic Operations with Complex Arrays

Remember that each of the operations mentioned in the previous section can be performed with complex arrays. The resulting array, if it is of type `COMPLEX`, will have both a real and an imaginary part in each element location. For example, you may have a two-dimensional complex array that looks like this:

$$Op\_array = \begin{pmatrix} 2 & 4 & -1 & 5 \\ -6 & 1 & 9 & 3 \end{pmatrix}$$

where the dimension statement is given as follows:

```
COMPLEX Op_array(-1:0,1:2)
```

The element `Op_array(-1,1)` contains the value:

```
2   4
```

where 2 is the real part of the complex number and 4 is the imaginary part.

If you were to multiply each of the complex values in the above matrix by a scalar value of 2, you would use the following statement:

```
MAT Complex_result= Op_array*(2)
```

The above statement would produce the following complex array:

$$Complex\_result = \begin{pmatrix} 4 & 8 & -2 & 10 \\ -12 & 2 & 18 & 6 \end{pmatrix}$$

Note that if the resulting array (`Complex_result`) had been of type `REAL` or `INTEGER`, the results in array `Complex_result` would look like this:

$$\begin{pmatrix} 4 & -2 \\ -12 & 18 \end{pmatrix}$$

This is due to the automatic type conversion made from `COMPLEX` to `REAL` or `INTEGER`. Notice that the imaginary part of the complex numbers in the array were dropped.

Summing the Elements in an Array

SUM is a function that returns the sum of *all* elements in an array. It works for arrays of any dimension. Given the array `A` below:

$$A = \begin{pmatrix} 4 & 2 & -1 \\ 3 & 8 & 16 \\ -5 & 2 & 0 \end{pmatrix}$$

The following use of the SUM function:

```
SUM(A)
```

would return 29.

There are also functions that compute the sum of an entire row or column of an array. However, these functions are limited to two-dimensional arrays and are discussed in the subsequent section of this chapter titled “Summing Rows and Columns of a Matrix.”

Boolean Arrays

In addition to the arithmetic operators, you can also use relational operators with arrays. The result is a boolean array (e.g., an array composed entirely of 1's and 0's, although—strictly speaking—these are not really boolean arrays since the values of the elements are not TRUE and FALSE). Given array B, suppose you wanted to know how many elements were greater than 50. First you execute the statement:

```
MAT F= B>(50)
```

which results in the array:

$$F = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

assuming array B has 4 elements in it greater than 50. Then you execute the statement,

```
PRINT SUM(F)
```

which causes the computer to display "4" on the current PRINTER IS device.

Note The *only* comparison operators allowed with COMPLEX arrays are = and <>.

You can also compare two arrays to each other. If, for example, you wanted to compare the two arrays below,

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 8 & 7 \\ 1 & 4 & 6 \end{pmatrix} B = \begin{pmatrix} 1 & 3 & 4 \\ 2 & 0 & 7 \\ 1 & 4 & 4 \end{pmatrix}$$

you could execute the statement:

```
MAT C= A=B
```

By looking at C, you can tell which elements are the same for both A and B

$$C = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Reordering Arrays

The `MAT REORDER` statement allows you to re-arrange an array so that one dimension is in a particular order. The new order is specified in a vector (a vector is a one-dimensional array). The vector contains the subscripts of the reordered dimension in their new order. The subscripts must correspond to the array's current dimensions and subscript ranges. Note that `MAT REORDER` works with `REAL`, `INTEGER`, `COMPLEX` and string arrays. However, as you might suspect, the reordering vector *cannot* be a `COMPLEX` vector.

Suppose `A` is the array below. Let us also assume that `A` has been dimensioned in `OPTION BASE 1`, and that the upper bound to both dimensions is 3.

$$A = \begin{pmatrix} 1 & 3 & 2 \\ 4 & 5 & 7 \\ 6 & 8 & 9 \end{pmatrix}$$

To reverse the order of the rows, we would first dimension a vector,

```
10 DIM Reverse(3)
```

and then assign its elements the following values:

```
Reverse(1)=3
Reverse(2)=2
Reverse(3)=1
```

The vector `Reverse` now contains:

$$Reverse = \begin{pmatrix} 3 & 2 & 1 \end{pmatrix}$$

If we execute the statement,

```
MAT REORDER A BY Reverse
```

the result array will be:

$$A = \begin{pmatrix} 6 & 8 & 9 \\ 4 & 5 & 7 \\ 1 & 3 & 2 \end{pmatrix}$$

Note that the rows are exchanged, rather than the columns. This is because the default is to re-order the 1st dimension. However, you can override the default by specifying a particular dimension to be re-ordered. For example, if we wanted to reverse columns rather than rows, we could use the same vector, but this time specify dimension 2:

```
MAT REORDER A BY Reverse,2
```

The transformation would be:

$$A = \begin{pmatrix} 6 & 8 & 9 \\ 4 & 5 & 7 \\ 1 & 3 & 2 \end{pmatrix} \Rightarrow A = \begin{pmatrix} 9 & 8 & 6 \\ 7 & 5 & 4 \\ 2 & 3 & 1 \end{pmatrix}$$

Remember that although our examples are confined to two dimensions for illustrative purposes, the same principles apply to arrays of three and more dimensions. In a three-dimensional array, for instance, reordering the 1st dimension would reorder planes rather than rows or columns.

In most cases, rather than creating a reorder vector and assigning values to it, you will already have a vector as the result of a sort operation. This is true except in the case of COMPLEX arrays which *cannot* be sorted because the < and > operators *are not* defined for them.

Sorting Arrays

Sorting an array rearranges the array so that one dimension (which you specify) is in numerical order. This section covers:

- Sorting with Automatic REORDER
- Sorting to a Vector

Note You cannot sort `COMPLEX` arrays because < and > operations are not defined for them.

Sorting with Automatic REORDER

Given the array A below, watch how the MAT SORT changes it.

$$\begin{array}{c}A\\ \begin{bmatrix}5&6&8\\3&5&1\\2&4&8\end{bmatrix}\end{array}$$

```
MAT SORT A(*,1)
```

$$\begin{array}{c}A\\ \begin{bmatrix}\boxed{2}&4&8\\\boxed{3}&5&1\\\boxed{5}&6&8\end{bmatrix}\end{array}$$

The asterisk specifies the dimension to be sorted, and the subscript(s) tells which elements in that dimension to use as the sorting values. In the example above, we told the system to sort rows (asterisk is located in the first subscript position), and to use the first element in each row as the sorting value. With the new array A (from the sort performed above), the following statement will sort columns using the second element in each column as the sorting value.

```
MAT SORT A(2,*)
```

$$\begin{array}{c}A\\ \begin{bmatrix}8&2&4\\\boxed{1}&\boxed{3}&\boxed{5}\\8&5&6\end{bmatrix}\end{array}$$

The key values in this sort are 1, 3 and 5, the second elements in each column. Sorting by placing the lowest values first is know as sorting by "ascending" order. This is the default. You can also sort by "descending" order by specifying the secondary keyword DES. The statement

```
MAT SORT A(*,2) DES
```

would produce the following transformation:

$$\begin{array}{c} A \\ \left[\begin{array}{ccc} 8 & 2 & 4 \\ 1 & 3 & 5 \\ 8 & 5 & 6 \end{array}\right] \end{array} \rightarrow \begin{array}{c} A \\ \left[\begin{array}{c|c|c} 8 & 5 & 6 \\ 1 & 3 & 5 \\ 8 & 2 & 4 \end{array}\right] \end{array}$$

Sometimes the values of two or more sorting elements are the same. For instance, if we sorted A by rows using the first element,

```
MAT SORT A(*,1)
```

we get:

$$\begin{array}{c} A \\ \left[\begin{array}{ccc} 8 & 5 & 6 \\ 1 & 3 & 5 \\ 8 & 2 & 4 \end{array}\right] \end{array} \rightarrow \begin{array}{c} A \\ \left[\begin{array}{|c|cc} 1 & 3 & 5 \\ 8 & 5 & 6 \\ 8 & 2 & 4 \end{array}\right] \end{array}$$

The first elements in the last two rows are the same, so the system leaves them in the order they held before the sort. However, you can specify a second sort element to be used in the case of ties. We could execute:

```
MAT SORT A(*,1),(*,2)
```

This tells the system to sort by rows using the first element as the sorting value; and in the case of ties, to use the second element. The result array would be as follows.

$$\begin{array}{c} A \\ \begin{pmatrix}1&3&5\\8&5&6\\8&2&4\end{pmatrix}\end{array} \rightarrow \begin{array}{c} A \\ \left(\begin{array}{c|c|c}1&3&5\\8&2&4\\8&5&6\end{array}\right)\end{array}$$

If a key is specified that is recognized by the system as rendering all other keys redundant (such as a non-substringed key for a one dimensional string array) no other keys can be specified. However, if the computer cannot tell that keys are redundant (such as `MAT SORT A(*,X),A(*,Y)` with `X` equal to `Y`) it will permit redundant keys. Redundant keys will slow down execution of the `MAT SORT` statement. If you include the `DES` secondary word, it refers only to the sort element which immediately precedes it.

Sorting to a Vector

So far, all of our sort examples have actually re-arranged the array in question. Alternatively, you can record the new order in a vector and leave the array intact. The vector must have been dimensioned to have at least as many elements as the current size of the array being sorted. If necessary, the system will redimension the vector. Thus, executing the statement:

```
MAT SORT A(3,*) TO Vect
```

with the array `A`:

$$A = \begin{pmatrix}1&3&5\\8&2&4\\8&5&6\end{pmatrix}$$

The array `A` remains unchanged, but the vector `Vect` now contains the values:

$$Vect = \begin{pmatrix}2&3&1\end{pmatrix}$$

This assumes that the array `A` has been dimensioned so that the subscript range is 1 thru 3. If `A` had been dimensioned:

```
10 DIM A(1:3,-1:1)
```

then `Vect` would contain the values:

$$Vect = \begin{pmatrix}0&1&-1\end{pmatrix}$$

This vector should look very reminiscent of the vectors used to reorder arrays. And, in fact, you can use these vectors in a MAT REORDER statement to rearrange the array. That is, we could now execute:

```
MAT REORDER A BY Vect,2
```

and the new array would be:

$$A = \begin{pmatrix} 3 & 5 & 1 \\ 2 & 4 & 8 \\ 5 & 6 & 8 \end{pmatrix}$$

Note that the dimension number in the MAT REORDER statement corresponds to the position of the asterisk in the MAT SORT statement.

Sorting to a vector is particularly useful if you want to sort the same array along different dimensions or using different sort elements. Each sort can be stored in a vector to be used later. Meanwhile, the original array remains unchanged.

In addition, sorting to a vector allows you to use the same sorting order with parallel arrays. That is, if you have several arrays that contain data about the same elements, you can sort one of them, and then use that same sorting order to reorder the others.

Finally, sorting to a vector enables you to manipulate an unsorted array as if it were sorted. For instance, suppose you have the array shown below.

$$A = \begin{pmatrix} 2 & 7 & 4 \\ 0 & 1 & 8 \\ 5 & 3 & 1 \end{pmatrix}$$

Let us also assume that the subscript range for each dimension in A is 1 thru 3. If we sort A to a vector B,

```
MAT SORT A(*,1) TO B
```

we can then use B to define elements in A. For instance to get the value of A(1,1) in its sorted form, we could write:

```
X=A(B(1),1)
```

In this case, X would equal 0. By incrementing the subscript value of B, we can simulate a sorted A array.

We should point out again that although these examples are two-dimensional, the same principles apply to arrays of any rank. You must have one, and only one, asterisk in the subscript list of a sort. The other subscripts specify the particular elements to be used as the sorting keys.

Searching Numeric Arrays

The purpose of the **MAT SEARCH** statement is to search for user-defined conditions within an array. This information is returned to a variable for recall and examination. Topics covered in this section are:

- Searching a vector
- Numeric comparisons in MAT SEARCH
- Searching a three-dimensional array
- Searching for multiple occurrences

For information on searching string arrays, read the chapter in the *HP BASIC Programming Guide* titled "String Manipulation."

Searching a Vector

The following program called `Mat_search` demonstrates a search for maximum and minimum values and their locations and the number of occurrences of the maximum and minimum values. It also includes a search for the location of a value less than a given expression. Note that within this program is a sample of some of the possible types of searches you can make using the **MAT SEARCH** statement. Lines 170 to 230 contain these sample searches.

```
100   OPTION BASE 1                        ! Select option base.
110   DIM Numbers(11)                      ! Dimension source array.
120   !
130   DATA 6,1,9,2,8,3,8,9,1,7,5           ! Random data.
140   !
150   READ Numbers(*)                      ! Input data to source array.
160   !
170   MAT SEARCH Numbers,MAX;Max           ! Search for maximum value.
180   MAT SEARCH Numbers,LOC MAX;Loc_max   ! Find location of maximum value.
```

```
190   MAT SEARCH Numbers,MIN;Min          ! Search for minimum value.
200   MAT SEARCH Numbers,LOC MIN;Loc_min  ! Find location of minimum value.
210   MAT SEARCH Numbers,#LOC(Max);Num_max   ! Search for # of maximums.
220   MAT SEARCH Numbers,#LOC(Min);Num_min   ! Search for # of minimums.
230   MAT SEARCH Numbers,LOC(<2);Loc_num,4  ! Starting with element 4,
240   !                                        return the first location of
250   !                                        a number less than 2.
270   ! Print the results.
280   !
290   PRINT "The maximum value is";Max;".";
300   PRINT " Its first occurrence is in array element";Loc_max;"."
310   PRINT
320   PRINT "The minimum value is";Min;"."; ! Print results.
330   PRINT " Its first occurrence is in array element";Loc_min;"."
340   PRINT
350   PRINT "The number of maximum value occurrences is";Num_max;"."
360   PRINT
370   PRINT "The number of minimum value occurrences is";Num_min;"."
380   PRINT
390   PRINT "Starting at array element 4, the first occurrence ";
400   PRINT "of a number"
410   PRINT "less than 2 is in array element";Loc_num;"."
420   END
```

If this program is run, the following results are obtained.

```
The maximum value is 9 . Its first occurrence is in array element 3 .

The minimum value is 1 . Its first occurrence is in array element 2 .

The number of maximum value occurrences is 2 .

The number of minimum value occurrences is 2 .

Starting at array element 4, the first occurrence of a number
less than 2 is in array element 9 .
```

Searching an Array by Descending Subscripts

If for some reason you need to search an array by descending subscript values, use the MAT SEARCH statement's DES option after the array's key specifier (i.e. Array(1,*) DES). This option causes a search to begin at the upper bound of a dimension in an array and proceed toward the lower bound of that same dimension. If a *starting subscript* is specified in the MAT SEARCH statement, then the search will begin at that specified location in the dimension being searched and proceeds toward the lower bound of that dimension. For example,

```
MAT SEARCH Array(1,*) DES,MAX;Max_value,6
```

searches the first row of a two-dimensional array called `Array` starting from the 6th element of that row and going toward the first element.

Before continuing with our discussion of searching an array by descending subscripts, let's look at the following table to clarify the difference between a search done by ascending subscripts and one done by descending subscripts.

Search Order	**Starting Subscript Given**	**No Starting Subscript Given**
ascending (default)	starting subscript → upper bound	lower bound → upper bound
descending	starting subscript → lower bound	upper bound → lower bound

If you substitute program lines 170 to 230 given below for the same program lines in the program called `Mat_search` explained in the previous section, you will find that the results produced are different.

```
170   MAT SEARCH Numbers DES,MAX;Max          ! Search for maximum value.
180   MAT SEARCH Numbers DES,LOC MAX;Loc_max  !Find location of maximum value.
190   MAT SEARCH Numbers DES,MIN;Min          ! Search for minimum value.
200   MAT SEARCH Numbers DES,LOC MIN;Loc_min  !Find location of minimum value.
210   MAT SEARCH Numbers DES,#LOC(Max);Num_max  ! Search for # of maximums.
220   MAT SEARCH Numbers DES,#LOC(Min);Num_min  ! Search for # of minimums.
230   MAT SEARCH Numbers DES,LOC(<2);Loc_num,4 ! Starting with element 4,
240   !                                          return the first location of
250   !                                          a number less than 2.
```

Executing the above program lines within the program called `Mat_search`, will result in the following output:

```
The maximum value is 9 . Its first occurrence is in array element 8 .

The minimum value is 1 . Its first occurrence is in array element 9 .

The number of maximum value occurrences is 2 .

The number of minimum value occurrences is 2 .

Starting at array element 4, the first occurrence of a number
less than 2 is in array element 2 .
```

Notice that the results given for the maximum and minimum values did not change and neither did the number of times they occurred within the output on the display. However, the locations of the data values did change. The reason for the change in data location is you began the search from the upper bound of the dimension being searched.

Numeric Comparisons in MAT SEARCH

Numeric comparisons are made when using the MAT SEARCH statement. The type of comparison made is determined by the *condition* option (e.g. MAX, LOC, MIN, etc.) of this statement. The MAX, MIN, LOC MAX, and LOC MIN conditions imply the use of the > and < relational operators (for MAX and MIN, respectively).

The *condition* options `LOC` and `#LOC` of a `MAT SEARCH` statement have as their arguments a relational operator along with an expression. There are six relational operators which can be used with this argument, they are: <, <= , =, <>, >=, >. If none of these operators are used, the default operator = is assumed. Some examples of these options being used to search a one-dimensional array are as follows.

`MAT SEARCH Array,LOC(<>4);Location`	assigns the location of the first element found not equal to 4 to the variable `Location`.
`MAT SEARCH Array,#LOC(>=7);Num_value`	assigns the total number of occurrences of values greater than or equal to 7 in the array called `Array` to the variable `Num_value`.

The expression is converted to the same data type as the array before the comparisons are done. For example, if **Array** is an INTEGER array, then the following statement:

```
MAT SEARCH Array,LOC(2.7);Location
```

assigns the *location* of the first occurrence of the value 3 (in **Array**) to the variable **Location**.

Note COMPLEX arrays can *only* be searched using the = and <> relational operators.

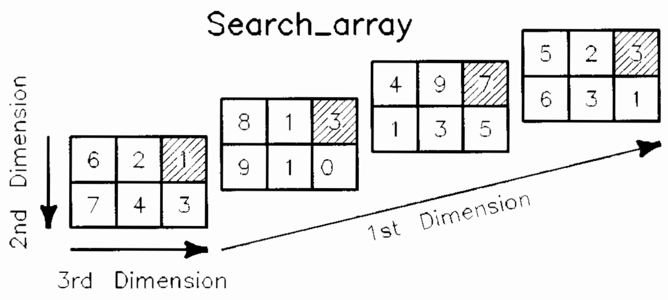
For a complete discussion of the *condition* option in the MAT SEARCH statement, consult the *HP BASIC Language Reference*.

Searching a Three-Dimensional Array

It is important to know that the MAT SEARCH statement works on only one dimension of an array. If you are searching a one-dimensional array there is no problem with searching the whole array. However, if you are searching a multi-dimensional array you need to specify only one dimension of that array in the *key specifier* of the MAT SEARCH statement. For example, assume that array **Search_array** is a three dimensional numeric array dimensioned using the following dimension statement:

```
100  DIM Search_array(4,2,3)
```

and that this array contains the following random numbers:



Assume that the shaded locations are to be searched until one is found which contains a value greater than 5. Let Value_loc represent the variable to which the location of the specified condition is returned.

The shaded memory locations must be described by a *key specifier*. Since they lie along the first dimension, the planes in which they lie are defined accordingly. The correct *key specifier* is:

```
(*,1,3)
```

Note that the *key specifier* is written in the same format as that used for a **MAT SORT** *key specifier*. The subscripts are written in the same order as the array dimensions (see the numbered arrows). The first subscript is an asterisk which indicates that the subscript is varied over its range of values, and the remaining subscripts define the fixed "row" and "column" locations.

The correct search statement for this example is

```
MAT SEARCH Search_array(*,1,3),LOC(>5);Value_loc
```

where:

Search_array	is the array being searched.
(*,1,3)	defines the locations to be searched.
LOC(>5)	specifies the condition to be satisfied.
Value_loc	is the variable to which the location value is returned.

After execution of the search statement, the number 3 is returned to the variable Value_loc. This is the first plane where the value of the specified location satisfies the condition option (i.e., it contains a value greater that 5). Searching begins at the lower bound of the *key specifier's* dimension which contains the asterisk, and proceeds toward the upper bound of the same dimension until a value satisfying the *condition* option is located.

If a condition is not satisfied upon completion of the searching process, a value one greater than the upper limit of the varied subscript is returned to the numeric variable. For example, if the specified locations in the previous example are searched for a value greater than 8, none is found. Therefore, a value of 5 representing a number one greater than the plane containing the last searched location is returned to Value_loc. Note that if the upper limit of the varied subscript is 32 767, the value returned by an unsuccessful search is −32 768.

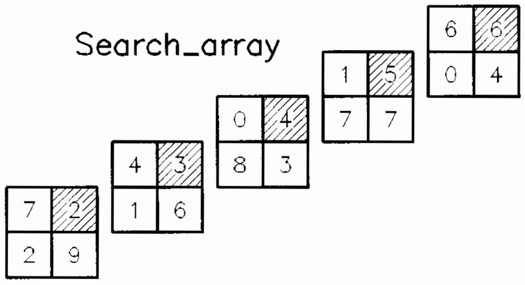
The *key specifier* initially defines the range of locations to be searched. If you do not wish to search the entire range, a *starting subscript* specifier can be used to designate where the search is to begin. Using the previous example, assume that the search process is to scan only the last three planes for the location of a value less than 5. The correct search statement to be used is

```
MAT SEARCH Search_array(*,1,3),LOC(<5);Value_loc,2
```

where the number 2 directs the search to begin at the specified location in the second plane and proceed to the last plane. Upon execution, the number 2 indicating the second plane is returned to the variable `Value_loc` because the content of its specified location is the first to satisfy the *condition*.

Searching for Multiple Occurrences

Normally, a `LOC` search ends at the first location which satisfies the specified condition. However, additional satisfactory values may exist beyond that location. By setting the starting address to be one greater than the content of the `Value_loc` variable, a search can be continued past the first location which satisfied the `LOC` condition. This automatically continues a search from where a previous search left off. All values which satisfy the given condition can be obtained in this way. As an example, assume that array `Search_array` is a three dimensional numeric array containing random numbers as shown.



Assume that the shaded memory locations are to searched for values greater than 2. A single `MAT SEARCH` scan would stop at the second plane since it is the first one encountered whose content satisfies the condition.

However, by constructing a loop and using the proper *starting subscript* specifier, the search can be made to continue through the range of specified locations. The following program demonstrates this feature.

```
100   OPTION BASE 1                ! Select the option base.
110   DIM Array(5,2,2)             ! Dimension Array.
120   !
130   DATA 7,2,2,9,4,3,1,6,0,4,8,3,1,5,7,7,6,6,0,4 ! Random data.
140   !
150   READ Array(*)                ! Read data into Array.
160   !
170   Subscript=0                  ! Initialize Subscript so first
180   !                              search is in plane 1 of Array.
190   !
200   LOOP
210     MAT SEARCH Array(*,1,2),LOC(>2);Subscript,Subscript+1 ! Search
220     !             for locations containing numbers greater than 2.
230     EXIT IF Subscript=6      ! Test: Have all locations been searched?
240     DISP Subscript;          ! Display most recent search results.
250     EXIT IF Subscript=5      ! Test: Has search reached last location?
260     !                              If not, continue.
270   END LOOP
280   END
```

The variable `Subscript` is initially set to zero. Lines 200 through 270 form a search loop. Line 210 begins the search routine. Since `Subscript` was set to zero, the starting subscript specifier (`Subscript + 1`) directs the search to begin at plane one. Line 230 tests to see if the specified locations have all been searched. (Remember, if all locations have been searched, a number one greater than the last plane searched is returned to the variable. In this case, that number is six.) If all locations have not yet been searched, line 240 displays the contents of `Subscript` (it now contains a satisfactory value). If `Subscript` contains a value less than 5, the search is not finished and line 270 directs the program to search again until a loop exit is made. Due to the nature of the starting subscript specifier, the search begins this time at the next memory location beyond that which satisfied the condition previously. In other words, the search resumes where it left off. If you run this program, the following should be displayed:

```
2  3  4  5
```

Matrices and Vectors

A two-dimensional numeric array is called a "matrix" and a one-dimensional numeric array is called a "vector". An entire branch of mathematics is devoted to matrices and vectors, and their applications are surprisingly broad. Keep in mind that the functions described in this section apply only to two-dimensional, and occasionally one-dimensional arrays, but never to arrays of more than two dimensions. Also, all functions described in this section apply to `REAL`, `INTEGER` and `COMPLEX` data types.

Matrix Multiplication

You may recall from our discussion of arrays and arithmetic operations that the asterisk (*) is reserved for matrix multiplication. If `A` is an *i-by-k* matrix and `B` is a *k-by-j* matrix, then $C=A*B$ is defined by the following equation:

$$C_{ij} = \sum_{n=1}^{k} A_{in} B_{nj}$$

Translated into English, this equation means that the element in the *ith* row and *jth* column of the product (`C`) is the sum of the products by pairs of the elements in the *ith* row of `A` and the *jth* column of `B`. A couple of examples will help make this clear.

Suppose `A` and `B` are the matrices shown below.

$$A = \begin{pmatrix} 3 & 8 & 2 \\ 1 & 6 & 5 \\ 4 & 2 & 0 \end{pmatrix} B = \begin{pmatrix} 1 & -2 & 3 \\ -6 & 4 & 7 \\ 0 & 8 & 2 \end{pmatrix}$$

```
MAT C= A*B
```

$$C = \begin{pmatrix} -45 & 42 & 69 \\ -35 & 62 & 55 \\ -8 & 0 & 26 \end{pmatrix}$$

Note that the product is a 3x3 matrix. There are three general rules to matrix multiplication:

- Multiplication between two matrices is legal only if the second dimension (k) of the first array is the same size as the first dimension of the second array. That is, the two inner dimensions must be the same.

- The result matrix will have the same number of rows as the first operand matrix and the same number of columns as the second operand matrix. That is, the dimensions of the result matrix will be the same as the outer dimensions of the operand matrices.
- The result array cannot be the same as either of the operand arrays. For example,

  ```
  MAT A= A*B
  ```

  is an illegal statement.

Suppose A is a 2 x 3 matrix and B is a 3 x 2 matrix, as shown below:

$$A = \begin{pmatrix} 6 & 8 & -1 \\ 2 & -3 & 4 \end{pmatrix} B = \begin{pmatrix} -1 & 1 \\ 2 & -2 \\ 3 & 4 \end{pmatrix}$$

Then:

$$A * B = \begin{pmatrix} 7 & -14 \\ 4 & 24 \end{pmatrix} B * A = \begin{pmatrix} -4 & -11 & 5 \\ 8 & 22 & -10 \\ 26 & 12 & 13 \end{pmatrix}$$

Note that matrix multiplication is *not* commutative, so `A*B≠B*A`.

Multiplication With Vectors

We described a vector as a one-dimensional array. For instance,

```
10  DIM A(3)
```

would create a vector with three elements and a rank of 1. Suppose we give `A` the values shown below.

$$A = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

Notice that we have portrayed `A` as a row vector. We could have just as easily portrayed `A` as a column vector:

$$A = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

So which is it? A row vector or a column vector? Actually, a vector can behave like either depending on its position in an equation. If a vector is the first operand in a multiplication, then it acts like an 1xn array (row vector); if it's the second operand, it behaves like a nx1 array (column vector); and if it's the result array, it can act like either. A few examples will help illustrate these principles. Let A be the vector shown above, and B, C, and D be the arrays shown below.

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} C = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} D = \begin{pmatrix} 2 & 2 & 2 \end{pmatrix}$$

Let us suppose that D has been explicitly defined as a two-dimensional array:

```
10 DIM D(3,1)
```

If we execute:

```
MAT C= A*D
```

we get:

$$C = \begin{pmatrix} 12 \end{pmatrix}$$

Since A is the first operand, it behaves like a 1x3 matrix. The equation, therefore, is:

$$C = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} * \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$

The result is a 1x1 matrix. If we try to reverse the order:

```
MAT C= D*A
```

the system returns:

```
ERROR 16 Improper dimensions
```

This is because we tried to multiply a 3x1 matrix by a 3x1 matrix:

$$C = \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Since the inner dimensions are not the same, the system returns an error. Suppose we try:

```
MAT C= B*A
```

In this case, we are multiplying a 3x3 array to a column vector:

$$C = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

The result is a 3×1 matrix:

$$C = \begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix}$$

If the result array is a vector, it will behave like either a row vector or a column vector depending on which is called for. The only other possibility is if both operand arrays are vectors. In this case, the result is always a 1×1 array. For instance, if `A` and `B` are vectors which are dimensioned as follows:

```
COMPLEX A(3),B(3)
```

and they contain the following complex values:

$$A = \begin{pmatrix} 2 & 1 \\ 4 & -1 \\ 6 & -2 \end{pmatrix} B = \begin{pmatrix} 0 & -3 \\ 1 & -6 \\ -1 & 5 \end{pmatrix}$$

then multiplying `A` by `B` results in a 1×1 array when the following equation is used.

```
MAT C= A*B
```

$$C = \begin{pmatrix} 2 \quad 1 & 4 \quad -1 & 6 \quad -2 \end{pmatrix} * \begin{pmatrix} 0 & -3 \\ 1 & -6 \\ -1 & 5 \end{pmatrix}$$

`C` equals `5 1`.

Note Complex arrays consist of pairs of numbers representing the real and imaginary components of each complex number, for example: `2 1`, `4 -1`, and `6 -2`.

Reversing the operand arrays, we get:

```
MAT C= B*A
```

$$C = \begin{pmatrix} 0 & -3 & 1 & -6 & -1 & 5 \end{pmatrix} * \begin{pmatrix} 2 & 1 \\ 4 & -1 \\ 6 & -2 \end{pmatrix}$$

Again, C equals 5 1.

Because the product of two vectors is always a single element, BASIC has a DOT function that multiplies two vectors and comes up with a REAL, INTEGER or COMPLEX numeric. For example,

```
X=DOT(A,B)
```

would assign the value 5 1 to X. If both vectors are INTEGER, then the product is INTEGER. If one is COMPLEX, the product is COMPLEX. Otherwise, the product is REAL. The two vectors must be the same size or the system will return an error.

Identity Matrix

An **identity matrix** is defined as a matrix which, when multiplied to another matrix A, produces the same matrix A. It is analogous to a 1 in normal arithmetic. For example, if I stands for an identity matrix, then A= I*A and also A= A*I. In order for an identity matrix to exist at all, A must be a square matrix (e.g., it must have the same number of columns as rows).

As it turns out, all identity matrices have the same form. They are square and consist of 1's along the main diagonal, and 0's everywhere else. For example, if A is a 3×3 matrix, then the identity matrix for A is:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Since identity matrices are used frequently in matrix arithmetic, BASIC has a special function (IDN) that turns a square matrix into an identity matrix.

```
10  OPTION BASE 1
20  COMPLEX I(2,2)
30  MAT I= IDN        I must be a square matrix
```

The COMPLEX matrix I now contains the elements:

$$I = \begin{pmatrix} 1 & 0 & & 0 & 0 \\ 0 & 0 & & 1 & 0 \end{pmatrix}$$

Inverse Matrix

Although division is not defined for matrices, there is a similar operation which involves finding the inverse of a matrix. A matrix must be square in order to have an inverse. If A is a square matrix, then A^{-1} denotes its inverse. The inverse is defined by the equation:

$$A * A^{-1} = I$$

where I is the identity matrix.

Note When using the inverse function (INV) if the source is INTEGER or REAL, then the destination *must be* REAL. If the source is COMPLEX, then the destination *must be* COMPLEX.

The inverse of a matrix is found by using the INV function. For instance, the inverse of:

$$A = \begin{pmatrix} 0 & 2 & 0 \\ -1 & 2 & 0 \\ 2 & 0 & 2 \end{pmatrix}$$

is found by executing the following statement.

```
MAT A_inv= INV(A)
```

BASIC computes the values of the inverse and places them in the matrix A_inv:

$$A\_inv = \begin{pmatrix} 1 & -1 & 0 \\ .5 & 0 & 0 \\ -1 & 1 & .5 \end{pmatrix}$$

To check that this is really the inverse, you could execute the statement:

```
MAT B= A*A_inv
```

As expected, B turns out to be an identity matrix:

$$B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Unfortunately, these expectations are not always fulfilled. Some matrices do not have an inverse. In other words, for a certain matrix called `A`, there exists no other matrix that, that when multiplied with (or by) `A` produces an identity matrix. Matrices that don't have an inverse are called **singular**. Singular matrices are easily detected. A more challenging type of matrix is one that is "ill-conditioned". Ill-conditioned matrices are ones whose inverse can't be found by the computer because of round-off errors. These are difficult to detect and almost impossible to correct. We'll talk more about singular and ill-conditioned matrices, but before we do, we should discuss why you'd use an inverse in the first place.

Solving Simultaneous Equations

One of the most common applications of matrices is in the solution of simultaneous equations. Simultaneous equations can be solved for `REAL`, `INTEGER` or `COMPLEX` data types.

Suppose we have the three equations shown below:

$$4X + 2Y - Z = 5$$

$$2X - 3Y + 3Z = 5$$

$$X + Y - 2Z = -3$$

We can re-write these equations in matrix format as the product of two arrays:

$$\begin{pmatrix} 4 & 2 & -1 \\ 2 & -3 & 3 \\ 1 & 1 & -2 \end{pmatrix} * \begin{pmatrix} \mathrm{X} \\ \mathrm{Y} \\ \mathrm{Z} \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \\ -3 \end{pmatrix}$$

For the sake of simplicity, let's name these three arrays `A`, `B`, and `C`. The equation, therefore, is:

$$A * B = C$$

If we multiply both sides of the equation by the inverse of A, we get:

$$A^{-1} * A * B = A^{-1} * C$$

Since $A^{-1} * A$ is simply a 3×3 identity matrix, the equation simplifies to:

$$I * B = A^{-1} * C$$

which further simplifies to:

$$B = A^{-1} * C$$

Remember, B is the matrix that contains the three variables X, Y *and* Z. To solve for these variables, therefore, all we have to do is multiply the matrix C by A^{-1}.

```
200   DIM Solution(3),A_inv(3,3)
 :
220   MAT A_inv= INV(A)
230   MAT Solution= A_inv*C
240   PRINT "X=";Solution(1)        X=1
250   PRINT "Y=";Solution(2)        Y=2
260   PRINT "Z=";Solution(3)        Z=3
```

For any set of simultaneous equations where there are the same number of unknown variables as there are equations, there are three possible classes of solution.

- There is no solution (e.g., there exist no values for the variables such that all of the equations are true).
- There are an infinite number of solutions.
- There is one, and only one, solution.

The first two cases are called "singular" sets of equations. You may recall that a singular matrix is one that has no inverse. It should not be surprising, therefore, that singular sets of equations always result in singular matrices when they are translated to matrix form. This is explained in the next section.

Singular Matrices

Any set of equations that has no solution or an infinite number of solutions is singular. Likewise, the matrix formed from these equations is also singular. More specifically, we mean the matrix on the left-hand side of the equation, what we've been calling matrix A. Consider the two equations listed below:

$$4X + 6Y = 5$$

$$4X + 6Y = 6$$

Obviously, there is no solution to this set of equations because any values assigned to X and Y will make only one of the equations true, not both. It is important to realize, however, that the singularity of these equations has nothing to do with the values on the right hand side of the equation. If, for example, we made the two equations the same,

$$4X + 6Y = 6$$

$$4X + 6Y = 6$$

then there would be an infinite number of solutions. For instance, X could equal 0 and Y equal 1, or X could equal 1.5 and Y could equal 0. In fact, so long as X=1.5(1-Y), the two equations will always be true. What is important here is that the two equations,

$$4X + 6Y =$$

$$4X + 6Y =$$

will be singular regardless of what we put on the right-hand side of the equal sign. If we translate these equations into matrix form, we get:

$$\begin{pmatrix} 4 & 6 \\ 4 & 6 \end{pmatrix} * \begin{pmatrix} X \\ Y \end{pmatrix}$$

The matrix,

$$\begin{pmatrix} 4 & 6 \\ 4 & 6 \end{pmatrix}$$

is singular: it has no inverse. If, however, we call this matrix A and do an INV on it, the system will not report an error. On the contrary, it will go ahead and find what it thinks is an inverse. However, whatever matrix it comes up with will not be the inverse. Let's see what happens with our singular matrix `A`.

```
MAT A_inv= INV(A)
PRINT A_inv(*)
```

When we execute these statements, the system will display the following:

```
.666666666667  .166666666667   0   -1
```

Arranging these values in the proper rows and columns, we get:

$$A\_inv = \begin{pmatrix} .666666666667 & 0 \\ .166666666667 & -1 \end{pmatrix}$$

To see whether this is a real inverse, we can multiply it by `A`. If it is the inverse, the product should be an identity matrix.

```
MAT I= A*A_inv
```

$$I = \begin{pmatrix} 3.33333333333 & 5 \\ -4 & -6 \end{pmatrix}$$

Obviously, the system has made a mistake—`A_inv` is not the inverse of `A`. So how do we know if an inverse is valid? Or, to put it another way, how do we detect a singular matrix? We have just seen one method: multiply the matrix by its inverse and see whether you get an identity matrix. There is, however, a much easier method. You simply look at the "determinant" of the matrix.

The Determinant of a Matrix

It's not really important that you understand how to calculate a determinant since the computer does it for you whenever you use the `DET` function. The `DET` function can be used with `REAL`, `INTEGER` and `COMPLEX` data types. For instance, to print the determinant of matrix `A`, you would write:

```
PRINT DET(A)
```

Also, the determinant is a byproduct of inversions. Thus, whenever you invert a matrix, the system computes the determinant and stores it. If you use `DET` without specifying a matrix, the system will return the determinant of the matrix most recently inverted. For example,

```
MAT A_inv= INV(A)
PRINT DET
```

would print the determinant of `A`.

If the determinant of a matrix equals 0, either a `REAL` underflow occurred during the inversion or *the matrix is singular*. To find out if an inversion is invalid, you merely test the matrix's determinant. If the determinant is zero, then the inverse is invalid. For example:

```
              .
              .
100   MAT A_inv= INV(A)       A must be square
110   IF DET=0 THEN Singular
              .
              .
```

If `A` is singular, program control is passed to a line named `Singular`. Note that we did not have to specify a matrix in line 110 since `A` was the last matrix inverted.

Unless you know for certain that a matrix is not singular, we recommend that you use the determinant test after each inversion. Otherwise, you may perform calculations using an invalid inverse.

Miscellaneous Matrix Functions

These functions are useful for obtaining the transpose of a matrix, summing the rows and columns of a matrix, and performing complex array operations. The topics covered are as follows:

- Transpose function
- Summing rows and columns of a matrix
- Examples of complex array operations

Transpose Function

The transpose of a matrix is derived by exchanging rows for columns and columns for rows. If **A** is the matrix below,

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
MAT B= TRN(A)
```

would result in:

$$B = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

A matrix does not have to be square to have a transpose. If **A** is,

$$A = \begin{pmatrix} 0 & 1 & 8 & 3 \\ 2 & 9 & 7 & -2 \end{pmatrix}$$

```
MAT B= TRN(A)
```

would result in:

$$B = \begin{pmatrix} 0 & 2 \\ 1 & 9 \\ 8 & 7 \\ 3 & -2 \end{pmatrix}$$

The result array cannot be the same as the array being transposed. For example,

```
MAT A= TRN(A)
```

is an illegal statement and will cause an error.

The transpose of a `COMPLEX` array is done in the same manner as for `REAL` and `INTEGER` values.

Summing Rows and Columns of a Matrix

BASIC has a function called RSUM which returns the sum of all rows in an array and a function called CSUM which returns the sum of all columns in an array. The totals are stored in a vector which RSUM and CSUM will re-dimension if necessary. Note that the DIM statement is needed in the following program because all other references to the arrays use (*) to specify the whole array. Let A be the matrix shown below.

$$A = \begin{pmatrix} 3 & 6 & 18 & 7 \\ 1 & 0 & 41 & 2 \\ 4 & 3 & 12 & 11 \end{pmatrix}$$

If we execute:

```
10    OPTION BASE 1
20    DIM A(3,4),Row_sum(3),Col_sum(4)
30    DATA 3,6,18,7,1,0,41,2,4,3,12,11
40    READ A(*)
50    MAT Row_sum= RSUM(A)
60    MAT Col_sum= CSUM(A)
70    PRINT "The sum of rows is: ";Row_sum(*)
80    PRINT "The sum of columns is: ";Col_sum(*)
90    END
```

BASIC will display:

```
The sum of rows is: 34 44 30
The sum of columns is: 8 9 71 20
```

Examples of Complex Array Operations

It is sometimes useful to create a REAL array from the real or imaginary parts of a COMPLEX array, as well as from the arguments or absolute values of each element of a COMPLEX array. It is also useful to be able to create a COMPLEX array from two REAL arrays. This section describes functions which allow you to perform these tasks. The COMPLEX array used in the examples is given below:

$$Complex\_array = \begin{pmatrix} -1 & -2 & & 3 & 5 & & -9 & 8 \\ 1 & 0 & & 2 & -7 & & 16 & -1 \end{pmatrix}$$

To place the real part of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= REAL(Complex_array)
```

which would result in the following array.

$$Array = \begin{pmatrix} -1 & 3 & -9 \\ 1 & 2 & 16 \end{pmatrix}$$

To place the imaginary part of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= IMAG(Complex_array)
```

which would result in the following array.

$$Array = \begin{pmatrix} -2 & 5 & 8 \\ 0 & -7 & -1 \end{pmatrix}$$

To place the argument of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= ARG(Complex_array)
```

which would result in the following array.

$$Array = \begin{pmatrix} -2.0344 & 1.0304 & 2.4150 \\ 0.0000 & -1.2925 & -.0624 \end{pmatrix}$$

Keep in mind that taking the ARG of an array returns values for the array elements which fall in the range of $-\pi$ to $+\pi$ for the radian mode and $-180°$ to $+180°$ for the degree mode.

To place the absolute value (or magnitude) of each element in a COMPLEX array called Complex_array into a REAL array called Array, you would use the following statement:

```
MAT Array= ABS(Complex_array)
```

which would result in the following array.

$$Array = \begin{pmatrix} 2.23617 & 5.8310 & 12.0416 \\ 1.0000 & 7.2801 & 16.0312 \end{pmatrix}$$

To create a conjugate array out of the COMPLEX array called Complex_array and place that conjugate array into the COMPLEX array called Conjugate, you would use the following statement:

```
MAT Conjugate= CONJG(Complex_array)
```

which would result in the following array.

$$Conjugate = \begin{pmatrix} -1 & 2 & 3 & -5 & -9 & -8 \\ 1 & 0 & 2 & 7 & 16 & 1 \end{pmatrix}$$

To create a COMPLEX array called New_comp_array from two REAL arrays called Real_array1 and Real_array2,

$$Real\_array1 = \begin{pmatrix} 4 & 6 & 7 \\ 5 & 9 & 1 \end{pmatrix} Real\_array2 = \begin{pmatrix} -4 & -8 & -1 \\ -3 & -2 & -9 \end{pmatrix}$$

you would use the following statement:

```
MAT New_comp_array= CMPLX(Real_array1,Real_array2)
```

which would result in the following array.

$$New\_comp\_array = \begin{pmatrix} 4 & -4 & 6 & -8 & 7 & -1 \\ 5 & -3 & 9 & -2 & 1 & -9 \end{pmatrix}$$

Using Arrays for Code Conversion

Suppose you have an input device that provides information in 8-bit ASCII code. On the other hand, an output device in the same system uses a non-ASCII specialized 8-bit code. Examples might include specialized instrumentation, typesetting equipment, or a multitude of other devices. For each ASCII character, there is a corresponding code for the output device. There may be some ASCII characters (such as control characters) that are not to be converted. Let us assume that a null character (all bits set to zero) is used for those special characters. Here is how a conversion array is set up:

1. First, an array is created with 256 elements (0 thru 255). Each element address corresponds to the 8-bit INTEGER numeric equivalent of the ASCII character code. The contents of a given array element contains the output code for the corresponding ASCII input code. The array can be REAL or INTEGER. Usually, it is more efficient to use INTEGER arrays for converting

16-bit or shorter codes. The array must be filled by individual program statements (assignments or DATA and READ statements), or it can be filled from a mass storage file. If a file is used, the data must be created by some prior means. Fixed conversion codes can sometimes be generated by an algorithm in the introductory part of the program that performs the conversions.

2. Input data is placed in a string variable (see the "String Manipulation" chapter in the *HP BASIC Programming Guide* for string variables techniques). Characters are then picked off, one character at a time, for conversion. Refer to the *HP BASIC Programming Guide* for more information about output operations.

Here is an example of how such an operation could be implemented:

```
1000   INTEGER Convert(0:255)
1010   DIM In$[80]
1020   Source=18 ! Source device selector
1030   Dest=22 ! Destination device selector
.
.
 Initialize the conversion array here.
.
.
2470   ENTER Source;In$ ! Input line of ASCII
2480   FOR I=1 TO LEN(In$) ! Send converted bytes
2490     OUTPUT Dest;CHR$(Convert(NUM(In$[I,I])));
2500   NEXT I
```

Note that the semicolon in line 2490 prevents sending a carriage-return and line-feed character pair at the end of each output line. This is usually necessary to prevent unwanted behavior when using ASCII strings to output non-ASCII data. This technique can be applied to arbitrary data conversions with virtually no limitations.

It is also possible to handle code conversions automatically in OUTPUT statements with the CONVERT options of the ASSIGN statement. See the ASSIGN Attributes discussion in the *HP BASIC Programming Guide*.

3 Strings and User-Defined Lexical Orders

Some applications may require special collating sequences. You can create a special lexical order as described in this chapter. However, for most applications you can use one of the standard, predefined lexical orders, which are discussed in chapter 5, "String Manipulation," in the *HP BASIC Programming Guide.*

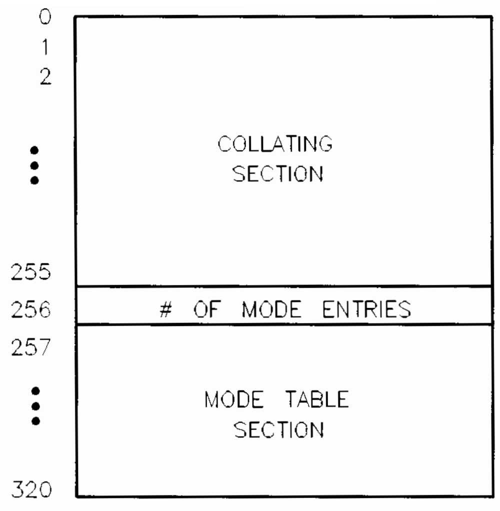
A program called LEX_AID has been supplied to simplify the creation of user-defined lexical orders. (For BASIC/WS this program is found on the *Utilities 2 Disk.* BASIC/UX installs the program in the /usr/lib/rmb/utils directory.) Before running the program it will be necessary to have an understanding of the terms used in this section. Using the LEX_AID program is described in the "BASIC Utilities Library" chapter of the *Installing and Maintaining HP BASIC* manual.

Basically, a 321 element (0 thru 320) INTEGER array is dimensioned, filled with sequence numbers and mode entries, and the new lexical order is established by the following statement.

```
LEXICAL ORDER IS Table(*)
```

Where Table(*) is any valid INTEGER array name.

The following illustration shows the general construction of a user-defined lexical table created in an INTEGER array.



The first 256 elements (0 through 255) contain the sequence number to be used in place of the character's ASCII value. For special characters, a mode type and mode table pointer are also stored in these elements.

The next element (256) contains the number of entries in the mode table. This value can range from 0 (no mode table) thru 64 (a full mode table).

The remaining 64 elements (257 thru 320) contain the optional mode table entries assigned to special characters.

Sequence Numbers

Normally, comparing two strings results in the computer comparing the ASCII values of the characters. When the computer makes the string comparison "A"<"B", the ASCII value of "A" (65) is compared to the ASCII value of the letter "B" (66) resulting in the comparison: 65<66, which is true.

Now suppose that a new value (sequence number) could be assigned to each of the ASCII characters. We might wish to assign the letter "A" a sequence number greater than the sequence number assigned to the letter "B". If such an assignment were made, the comparison "A"<"B", would now be false.

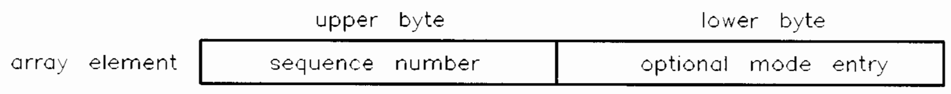
Once a lexical order is invoked, if two strings are compared, the strings are first converted into two series of sequence numbers and the comparison is then based on the sequence numbers.

The `LEXICAL ORDER IS` statement's primary purpose is to assign a sequence number to each character. However, this is not always enough to handle certain character combinations and special cases encountered in other languages. Special characters have a mode entry included with the sequence number.

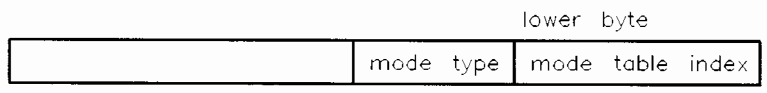
Mode Entries

Each of the first 256 array elements (0 thru 255) contains the sequence number to be used in place of the character's ASCII value. Optionally, a mode entry can be included.

Internally, an integer array element uses two bytes (16 bits) of memory. In the following diagram, the array element is divided into its upper, and lower bytes. The upper byte contains the sequence number and the lower byte is used if the character has a mode entry.



The lower byte is further divided into two parts. The upper-most 2-bits are used to represent one of the four mode types. The remaining 6-bits store an index (pointer) to the actual mode table entries. This method allows all the necessary information, for each character, to be stored as a single element in the INTEGER array.



Mode Type

Any one of the following mode types can be assigned to a character.

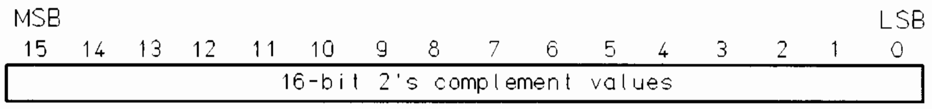
- Don't Care Characters (Mode type: 0)
- "1 for 2" Character Replacements (Mode type: 1)
- "2 for 1" Character Replacements (Mode type: 2)
- Accent Priority (Mode type: 3)

Mode Index

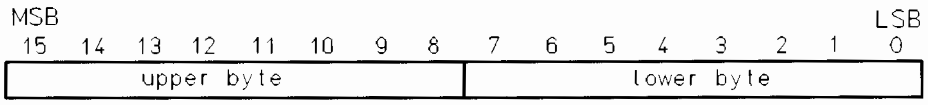
The mode index points to the actual mode table entry associated with the particular character. Up to 64 indexes are allowed (0 thru 63); however, some mode types use more than one table entry.

Bits, Bytes, and Mode Types

Each INTEGER array element stores a signed-integer in the range: −32768 thru 32767. Internally, the number is stored as a 16-bit 2's complement value, as follows:



However, we want to store one of 256 possible sequence numbers and, optionally, a mode type and mode table index. Since there are 256 characters used with the `LEXICAL ORDER IS` statement, and 8 bits are needed to store one of 256 possible values ($2^8 = 256$), it is convenient to think of the bits arranged as two bytes (a byte contains 8 bits), as follows:



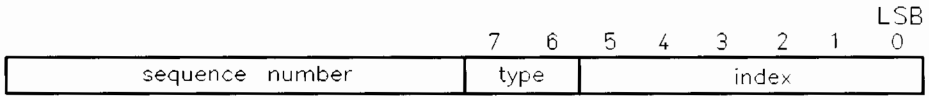
The upper byte is used to hold the sequence number and the lower byte contains the mode entry information. The algorithm below will produce a signed 16-bit integer from two unsigned 8-bit bytes.

```
Integer = (256*Upper+Lower)-(Upper>127)*65536
```

The process can be reversed.

```
IF Integer<0 THEN Integer=Integer+65536
Upper = Integer DIV 256
Lower = Integer MOD 256
```

The lower byte is further divided into two groups. Two bits hold one of four mode types (2^2=4) and the remaining six bits are for one of 64 mode indexes (2^6=64).



A "1 for 2" entry is signified by bit-6 being set. Therefore the value of the lower byte can range from 64 thru 126. (a "1 for 2" requires at least 2 entries.)

A "2 for 1" entry has bit-7 set. The value of the lower byte can range from 128 thru 191.

An "Accent priority" entry has both bit-6 and bit-7 set. The value of the lower byte ranges from 192 thru 255.

"Don't Care" Characters

A character can be removed from the collation sequence. To mark a character as a "don't care", the mode type is 0 (the same as a regular character) but the mode table index is set to 1.

sequence number	type	index
any value	0	1

The mode index need not point to a valid table entry, but must be a "1" to indicate a "don't care" character.

For example, the FRENCH lexical table lists the hyphen (-) as a "don't care" character. Thus, the hyphen is ignored when a string comparison is being made. The entry appears:

	sequence number	type	index
(45)	45	0	1

You may wish to include "don't care" characters in your own lexical tables. A string containing only "don't care" characters will match the null string.

The following short program illustrates the operation of a "don't care" character.

```
10 Return$="RESTORE"
20 Again$="RE-STORE"
30 !
40 LEXICAL ORDER IS ASCII
50 IF Restore$=Again$ THEN PRINT "True for ASCII"
60 LEXICAL ORDER IS FRENCH
70 IF Restore$=Again$ THEN PRINT "True for FRENCH"
80 END
```

Result:

```
True for FRENCH
```

"1 for 2" Character Replacement

This type of mode table entry indicates that one sequence number is to be used for two consecutive characters. It should be remembered that no characters are actually replaced by this operation, only that a single sequence number is to be used when the two characters are found adjacent to each other.

The following entry is placed in the collating section of the lexical table.

sequence number	type	mode index
normal sequence number	1	index

If a character marked as a "1 for 2" is found in a string, the next character is accessed and compared to the list of possible secondary characters in the mode table section.

(257 + index)	number of entries to check	
	second character	sequence number of this pair
	second character	sequence number of this pair

If the character does not match any of the secondary characters in the mode table, the original character's sequence number is used and processing continues. If a match is found, the sequence number for the pair is used and processing continues with the character following the secondary character.

For example, the SPANISH collating sequence has a "1 for 2" replacement for the letters "CH" or "Ch". The letter "C" is marked as a "1 for 2" character. When the letter is encountered in a string, the next character is accessed and compared to the list of possible secondary characters (upper-case H and lower-case h). The appropriate sequence number is then used for the pair. If the character following the letter "C" is not found in the list of possible secondary characters, the sequence number for "C" is used and processing continues with the next character.

You can override a “1 for 2” character replacement by inserting a “Don’t Care” character between the two characters that would otherwise be replaced by a single sequence number.

The SPANISH table entry for the character sequence “CH” is below. In the collating section, the first letter of the sequence has the following entry:

	sequence number	type	mode index
(67)	67 (C)	1	1
(68)	68 (D)	0	0

(257 + 1)	number of entries to check (2)	
(257 + 2)	second character (H)	sequence number for pair (68)
(257 + 3)	second character (H)	sequence number for pair (68)

The sequence number assigned to the two-character combination is greater than the sequence number for the letter “C” and less than the sequence number for the letter “D”. Therefore, a word beginning with the characters “CH” will collate after all words starting with the letter “C” followed by any other character.

The following program shows the sorting order for the letters “CH” in the SPANISH lexical order.

```
5 DIM A$(3)[3]
10 A$(1)="CGA"
20 A$(2)="CHA"
30 A$(3)="CIA"
40 LEXICAL ORDER IS SPANISH
50 MAT SORT A$(*)
60 PRINT A$(*)
70 END
```

Produces:

```
CGA   CIA   CHA
```

It should be noted that a character may have more than one secondary character combination. This is demonstrated by having both upper and lower case entries. Other secondary characters could have been included in the same manner. The first mode table entry contains the number of secondary characters to check and must be in the range: 0 thru 63.

"2 for 1" Character Replacement

When a "2 for 1" mode entry is specified, it indicates that the character should be represented by two sequence numbers (as if there were two characters in the string). The first sequence number is stored with the character as usual. The mode index points to the mode table entry that contains the second sequence number to be used for that character.

sequence number	type	mode index
1st sequence number	2	index

The mode table entry actually contains two sequence numbers. If the original character was upper case, the next character in the string will determine whether the upper or the lower sequence number is used. If the next character in the string is upper-case, the upper sequence number is used as the second sequence number. Conversely, if the next character in the string is lower-case, the lower sequence number is used. If the original character was a lower-case letter, the lower sequence number is always used.

	upper	lower
(257 + index)	2nd sequence number (UPC)	2nd sequence number (LWC)

Several "2 for 1" characters are in the GERMAN lexical order. For instance, the character "Ä" is equivalent to "AE" and has the following entry in the collating section.

	sequence number	type	mode index
(216)	65 (A)	2	index

The index points to the following entry in the mode table.

	upper	lower
(257 + index)	75 (E)	124 (e)

In some cases, such as the character “ß”, both upper and lower bytes contain the sequence number for the same character (s). This results in the same sequence numbers being generated regardless of the case of the next character.

Accent Priority

Accent Priority can be used as the final arbitrator of string comparisons. If you examine the lexical tables you will often find the same sequence number assigned to more than one character. Therefore, it is possible for two different strings to produce identical series of sequence numbers. The two strings will be considered equal unless at least one character, in each string, has been assigned different accent priorities.

Accent priority is established by assigning a value, in the range: 0 thru 63, to the character. Any character not already assigned a mode type may be assigned a priority. A priority of zero is assumed for all characters that haven’t been assigned a priority.

sequence number	type	mode index
normal sequence number	3	priority

In the FRENCH lexical order, the characters: A, Á, and À have been assigned the same sequence number (64). Assume the characters were assigned the following priorities.

Character	**Priority**	
A	0	(default priority)
Á	1	
À	2	

The characters can now be distinguished from one another and will collate in the following order.

```
A < Á < À
```

When two strings are compared, each string is first converted into a series of sequence numbers. The comparison is then determined (in most cases) by the greater sequence numbers or the longer series of sequence numbers.

In the event both strings produce identical series of sequence numbers, the series of priorities are checked. The string containing the characters with the higher priority is the greater string.

A Closer Look at File I/O

This chapter describes selected file I/O topics for the advanced programmer, including a detailed discussion of file pointers. For a general discussion of file I/O, refer to chapter 7, "Data Storage and Retrieval," in the *HP BASIC Programming Guide*.

Using ASCII Files

Chapter 7 in the *HP BASIC Programming Guide* describes the various types of BASIC data files (ASCII, BDAT, HP-UX, and DOS), and tells how to choose a file type. Let's take a closer look at the file I/O techniques for ASCII files.

Example of ASCII File I/O

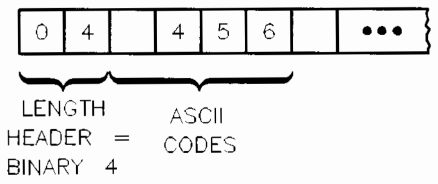
Storing data in ASCII files requires a few simple steps. The following program segment shows a simple example of placing several items in an ASCII data file.

```
100  REAL Real_array1(1:50,1:25),Real_array2(1:50,1:25)
110  INTEGER Integer_var
120  DIM String$[100]
       .
       .
390  ! Specify "default" mass storage device.
400  MASS STORAGE IS ":,700,1"
410  !
420  ! Create ASCII data file with 10 sectors
430  ! on the "default" mass storage device.
440  CREATE ASCII "File_2",10
450  !
460  ! Assign (open) an I/O path name to the file.
470  ASSIGN @Path_1 TO "File_2"
480  !
```

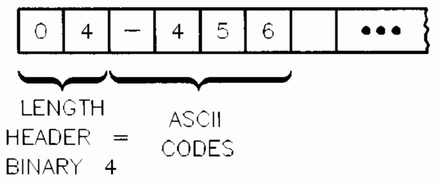
```
490  ! Write various data items into the file.
500  OUTPUT @Path_1;"Literal"       ! String literal.
510  OUTPUT @Path_1;Real_array1(*)  ! REAL array.
520  OUTPUT @Path_1;255             ! Single INTEGER.
530  !
540  ! Close the I/O path.
550  ASSIGN @Path_1 TO *
       .
       .

790  ! Open another I/O path to the file (assume same default drive).
800  ASSIGN @F_1 TO "File_2"
810  !
820  ! Read data into another array (same size and type).
830  ENTER @F_1;String_var          ! Must be same data types.
840  ENTER @F_1;Real_array2(*)
850  ENTER @F_1;Integer_var
860  !
       .
       .

870  ! Close I/O path.
880  ASSIGN @F_1 TO *
       .
       .
```

Data Representations in ASCII Files

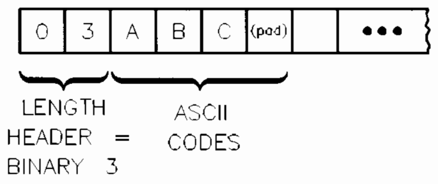
In an ASCII file, every data item, whether string or numeric, is represented by ASCII characters; one byte represents one ASCII character. Each data item is preceded by a two-byte length header which indicates how many ASCII characters are in the item. However, there is no "type" field for each item; data items contain no indication (in the file) as to whether the item was stored as string or numeric data. For instance, the number 456 would be stored as follows in an ASCII file:



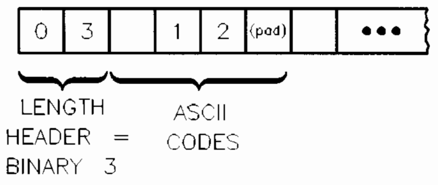
Note that there is a space at the beginning of the data item. This signifies that the number is positive. If a number is negative, a minus sign precedes the number. For instance, the number −456, would be stored as follows:



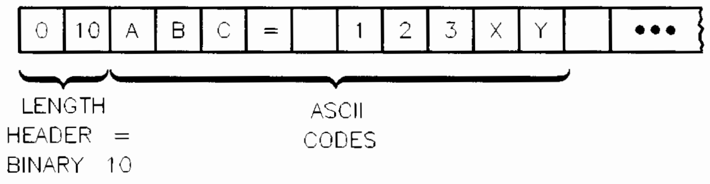
If the length of the data item is an odd number, the system "pads" the item with a space to make it come out even. The string "ABC", for example, would be stored as follows:



There is often a relatively large amount of overhead for numeric data items. For instance, to store the integer 12 in an ASCII file requires the following six bytes:



Similarly, reading numeric data from an ASCII file can be a complex and relatively slow operation. The numeric characters in an item must be entered and evaluated individually by the system's "number builder" routine, which derives the number's internal representation. (Keep in mind that this routine is called automatically when data are entered into a numeric variable.) For example, suppose that the following item is stored in an ASCII file:



Although it may seem obvious that this is not a numeric data item, the system has no way of knowing this since *there is no type-field stored with the item.* Therefore, if you attempt to enter this item into a numeric variable, the system uses the number-builder routine to strip away all non-numeric characters and spaces and assign the value 123 to the numeric variable. When you add to this the intricacies of real numbers and exponential notation, the situation becomes more complex. For more information about how the number builder works, see the chapter called "Entering Data" in the *HP BASIC Programming Guide.*

Because ASCII files require so much overhead (for storage of "small" items), and because retrieving numeric data from ASCII files is sometimes a complex process, they are not the preferred file type for numeric data when compactness is an important criteria. However, ASCII files are interchangeable with many other HP products.

In this chapter, we refer to the data representation described above as ASCII-data format. You can also store data in BDAT files in ASCII format (by using the FORMAT ON attribute). Be careful not to confuse the ASCII-*file type* with the ASCII-*data format.* The ASCII format used in BDAT files when FORMAT ON is specified differs from the format used in ASCII files in several respects. Each item output to an ASCII file has its own length header; there are no length headers in a FORMAT ON BDAT file. At the end of each OUTPUT statement an end-of-line sequence is written to a FORMAT ON BDAT file unless suppressed by an IMAGE or EOL OFF. No end-of-line sequence is written to an ASCII file at the end of an OUTPUT statement.

In general, you should only use ASCII files when you want to transport data between HP Series 200/300/400/700 computers and other HP computers. There may be other instances where you will want to use ASCII files, but you should be aware that they cause a *noticeable transfer rate degradation* compared to BDAT and HP-UX files (especially for numeric data items).

Formatted OUTPUT with ASCII Files

As mentioned in the "Brief Comparison of File Types," you cannot format items sent to ASCII files; that is, you *cannot* use the following statement with an ASCII file:

```
OUTPUT @Ascii_file USING "#,DD.D,4X,5A";Number,String$
```

You can, however, direct the output to a string variable first, and then OUTPUT this formatted string to an ASCII file:

```
OUTPUT String_var$ USING "#,DD.D,4X,5A";Number,String$
OUTPUT @Ascii_file;String_var$
```

When a string variable is specified as the destination of data in an OUTPUT statement, source items are evaluated individually and placed into the variable according to the free-field rules or the specified image, depending on which type of OUTPUT statement is used. Thus, item terminators may or may not be placed into the variable. The ASCII data representation is always used during outputs to string variables; in fact, *data output to string variables is exactly like that sent to devices through I/O paths with the FORMAT ON attribute.*

When using OUTPUT to a string, characters are always placed into the variable beginning at the first position; no other position can be specified as the beginning position at which data will be placed. Thus, *random access of the information in string variables is not allowed* from OUTPUT and ENTER statements; all data must be accessed serially. For instance, if the characters "1234" are output to a string variable by one OUTPUT statement, and a subsequent OUTPUT statement outputs the characters "5678" to the same variable, the second output *does not* begin where the first one left off (i.e., at string position five). The second OUTPUT statement begins placing characters in position one, just as the first OUTPUT statement did, overwriting the data initially output to the variable by the first OUTPUT statement.

The string variable's length header (2 bytes) is updated and compared to the dimensioned length of the string as characters are output to the variable. If the string is filled before all items have been output, an error is reported; however, the string contains the first *n* characters output (where *n* is the dimensioned length of the string).

4

Example

The following program outputs string and numeric data items to a string variable and then calls a subprogram which displays each character, its decimal code, and its position within the variable. Even though this program does not write to an ASCII file it shows a character representation of what would appear in an ASCII file.

```
100   ASSIGN @Crt TO 1  ! CRT is disp. device.
110   !
120   OUTPUT Str_var$;12,"AB",34
130   !
140   CALL Read_string(@Crt,Str_var$)
150   !
160   END
170   !
180   !
190 SUB Read_string(@Disp,Str_var$)
200     !
210     ! Table heading.
220     OUTPUT @Disp;"---------------------"
230     OUTPUT @Disp;"Character  Code  Pos."
240     OUTPUT @Disp;"---------  ----  ----"
250     Dsp_img$="2X,4A,5X,3D,2X,3D"
260     !
270     ! Now read the string's contents.
280   FOR Str_pos=1 TO LEN(Str_var$)
290         Code=NUM(Str_var$[Str_pos;1])
300       IF Code<32 THEN ! Don't disp. CTRL chars.
310           Char$="CTRL"
320       ELSE
330           Char$=Str_var$[Str_pos;1] ! Disp. char.
340       END IF
350       !
360       OUTPUT @Disp USING Dsp_img$;Char$,Code,Str_pos
370   NEXT Str_pos
380   !
390   ! Finish table.
400   OUTPUT @Disp;"---------------------"
410   OUTPUT @Disp ! Blank line.
420   !
430   SUBEND
```

```
 --------------------
 Character  Code  Pos.
 ---------  ----  ----
             32    1
   1         49    2
   2         50    3
   ,         44    4
   A         65    5
   B         66    6
   CTRL      13    7
   CTRL      10    8
             32    9
   3         51    10
   4         52    11
   CTRL      13    12
   CTRL      10    13
 --------------------
```

Outputting data to a string and then examining the string's contents is usually a more convenient method of examining output data streams than using a mass storage file. The preceding subprogram may facilitate the search for control characters. They are not displayed, because they might cause the printer or CRT to perform control actions.

The following example program shows how outputs to string variables can be used to reduce the overhead required in ASCII data files. To do this, the program compares two possible methods for storing data in an ASCII data file. The first method stores 64 two-byte items in a file one at a time. Each two-byte item is preceded by a two-byte length header. The second method stores 64 two-byte items in a string array which is output to a string variable. The string variable is then output to an ASCII data file with only one two-byte length header being used. Since the second method used only one two-byte length header to store 64 two-byte items, it can easily be seen that the second method required less overhead. Note that the second method is also the *only way to format data sent to ASCII data files.*

```
100   PRINTER IS CRT
110   !
120   ! Create a file 1 record long (=256 bytes).
130   ON ERROR GOTO File_exists
```

```
140   CREATE ASCII "TABLE",1
150 File_exists:   OFF ERROR
160                !
170                !
180   ! First method outputs 64 items individually..
190   ASSIGN @Ascii TO "TABLE"
200   FOR Item=1 TO 64  ! Store 64 2-byte items.
210       OUTPUT @Ascii;CHR$(Item+31)&CHR$(64+RND*32)
220       STATUS @Ascii,5;Rec,Byte
230       DISP USING Image_1;Item,Rec,Byte
240   NEXT Item
250 Image_1: IMAGE "Item ",DD,"  Record ",D,"  Byte ",3D
260   DISP
270   Bytes_used=256*(Rec-1)+Byte-1
280   PRINT Bytes_used;" bytes used with 1st method."
290   PRINT
300   PRINT
310   !
320   !
330   ! Second method consolidates items.
340   DIM Array$(1:64)[2],String$[128]
350   ASSIGN @Ascii TO "TABLE"
360   !
370   FOR Item=1 TO 64
380       Array$(Item)=CHR$(Item+31)&CHR$(64+RND*32)
390   NEXT Item
400   !
410   OUTPUT String$;Array$(*); ! Consolidate in string variable.
420   OUTPUT @Ascii;String$     ! OUTPUT to file as 1 item.
430   !
440   STATUS @Ascii,5;Rec,Byte
450   Bytes_used=256*(Rec-1)+Byte-1
460   PRINT Bytes_used;" bytes used with 2nd method."
470   !
480   END
```

The program shows many of the features of using ASCII files and string variables. The first method of outputting the data items shows how the file pointer varies as data are sent to the file. Note that the file pointer points to the *next* file position at which a subsequent byte will be placed. In this case, it is incremented by four by every OUTPUT statement (since each item is a two-byte quantity preceded by a two-byte length header).

The program could have used a BDAT file, which would have resulted in using slightly less disk-media space; however, using BDAT files usually saves much more disk space than would be saved in this example. The program does not show that *ASCII files cannot be accessed randomly*; this is one of the major differences between using ASCII and BDAT (and HP-UX) files.

Formatted ENTER with ASCII Files

Data is entered from string variables in much the same manner as output to the variable. For example,

```
ENTER @File;String$
ENTER String$;Var1, Var2$
```

All ENTER statements that use string variables as the data source interpret the data according to the FORMAT ON attribute. Data is read from the variable beginning at the first string position; if a subsequent ENTER statement reads characters from the variable, the read also begins at the first position. If more data is to be entered from the string than is contained in the string, an error is reported; however, all data entered into the destination variable(s) before the end of the string was encountered remain in the variable(s) after the error occurs.

When entering data from a string variable, the computer keeps track of the number of characters taken from the variable and compares it to the string length. Thus, statement-termination conditions are *not* required; the ENTER statement automatically terminates when the last character is read from the variable. However, *item* terminators are still required *if* the items are to be separated *and* the lengths of the items are not known. If the length of each item is known, an image can be used to separate the items.

Example

The following program shows an example of the need for *either* item terminators *or* length of each item. The first item was not properly terminated and caused the second item to not be recognized.

```
100   OUTPUT String$;"ABC123";  ! OUTPUT w/o CR/LF.
110   !
120   ! Now enter the data.
130   ON ERROR GOTO Try_again
140   !
```

```
150 First_try: !
160   ENTER String$;Str$,Num
170   OUTPUT CRT;"First try results:"
180   OUTPUT CRT;"Str$= ";Str$,"Num=";Num
190   BEEP     ! Report getting this far.
200   STOP
210   !
220 Try_again: OUTPUT CRT;"Error";ERRN;" on 1st try"
230            OUTPUT CRT;"STR$=";Str$,"Num=";Num
240            OUTPUT CRT
250            OFF ERROR  ! The next one will work.
260            !
270   ENTER String$ USING "3A,3D";Str$,Num
280   OUTPUT CRT;"Second try results:"
290   OUTPUT CRT;"Str$= ";Str$,"Num=";Num
300   !
310   END
```

Results:

```
Error 153 on 1st try
Str$=ABC123
Num= 0

Second try results:
Str$= ABC
Num= 123
```

This technique is convenient when attempting to enter an unknown amount of data or when numeric and string items within incoming data are not terminated. The data can be entered into a string variable and then searched by using images.

Example

ENTERs from string variables can also be used to generate a number from ASCII numeric characters.

```
30    Number$="Value= 43.5879E-13"
40    !
50    ENTER Number$;Value
60    PRINT "VALUE=";Value
70    END
```

Example

An ASCII file can always be read as strings even if the data is numeric.

```
10    DIM Buf$[80]               Read any ASCII files, line ≤ 80 characters
20    ASSIGN @File TO "File"
30    ON END @File GOTO Ending
40    LOOP
50     ENTER @File;Buf$
60     PRINT Buf$
70    END LOOP
80 Ending: !
90    END
```

A Closer Look at BDAT and HP-UX Files

As mentioned earlier, BDAT and HP-UX files are designed for flexibility (random and serial access, choice of data representations), storage-space efficiency, and speed. This chapter provides several examples of using these types of files.

Data Representations Available

The data representations available are:

- BASIC internal formats (allow the fastest data rates and are generally the most space-efficient)
- ASCII format (the most interchangeable)
- Custom formats (design your own data representations using IMAGE specifiers)

More details of each type of representation are described in the remainder of this section.

Random vs. Serial Access

Random access means that you can directly read from and write to any record within the file, while serial access only permits you to access the file in order, from the beginning. That is, you must read records 1, 2, ... , $n-1$ before you can read record n. Serial access can waste a lot of time if you're trying to access data at the end of a file. On the other hand, if you want to access the entire file sequentially, you are better off using serial access than random access, because it generally requires less programming effort and often uses less file space. BDAT and HP-UX files can be accessed both randomly and serially, while ASCII files can be accessed only serially.

Data Representations Used in BDAT Files

BDAT files allow you to store and retrieve data using internal format, ASCII format, or user-defined formats.

- With internal format (FORMAT OFF), items are represented with the same format the system uses to store data in internal computer memory. (This is the default FORMAT for BDAT and HP-UX files.)
- With ASCII format (FORMAT ON), items are represented by ASCII characters.
- User-defined formats are implemented with programs that employ OUTPUT and ENTER statements that reference IMAGE specifiers (items are represented by ASCII characters).

This section shows the details of internal (FORMAT OFF) representations of numeric and string data. Complete descriptions of ASCII and user-defined formats are given in the "Introduction to I/O" chapter of the *HP BASIC Programming Guide*.

BDAT Internal Representations (FORMAT OFF)

In most applications, you will use internal format for BDAT files. Unless we specify otherwise, you can assume that when we talk about retrieving and storing data in BDAT files, we are also talking about internal format. This format is synonymous with the FORMAT OFF attribute, which is described later in this chapter.

Because FORMAT OFF assigned to BDAT files uses almost the same format as internal memory, very little interpretation is needed to transfer data between the computer and a FORMAT OFF file. FORMAT OFF files, therefore, not only save space but also save time.

Data stored in internal format in BDAT files require the following number of bytes per item:

Data Type	**Internal Representation**
INTEGER	2 bytes
REAL	8 bytes
COMPLEX	16 bytes (same as 2 REALs)
String	4-byte length header; 1 byte per character (plus 1 pad byte if string length is an odd number)

INTEGER values are represented in BDAT files which have the FORMAT OFF attribute by using a 16-bit, two's-complement notation, which provides a range −32 768 thru 32 767. If bit 15 (the MSB) is 0, the number is positive. If bit 15 equals 1, the number is negative; the value of the negative number is obtained by changing all ones to zeros, and all zeros to ones, and then adding one to the resulting value.

Examples

Binary Representation	Decimal Equivalent
00000000 00010111	23
11111111 11101000	−24
10000000 00000000	−32768
01111111 11111111	32767
11111111 11111111	−1
00000000 00000001	1

REAL values are stored in BDAT files by using their internal format (when FORMAT OFF is in effect): the IEEE-standard, 64-bit, floating-point notation. Each REAL number is comprised of two parts: an exponent (11 bits), and a mantissa (53 bits). The mantissa uses a sign-and-magnitude notation. The sign bit for the mantissa is not contiguous with the rest of the mantissa bits; it is the most significant bit (MSB) of the entire eight bytes. The 11-bit exponent is offset by 1 023 and occupies the 2nd through the 12th MSB's. Every REAL number is internally represented by the following equation. (Note that the mantissa is in binary notation):

$$-1^{mantissasign} \times 2^{exponent-1023} \times 1.mantissa$$

Thus, the real number 1/3 would be stored as:

```
byte 1   byte 2   byte 3   byte 4   ... byte 8
00111111 11010101 01010101 01010101 ... 01010101
```

COMPLEX values are always stored as two REAL values.

String data are stored in FORMAT OFF BDAT files in their internal format.

- A 4-byte length header contains a value that specifies the length of the string (the 2 leading bytes of length header are always 0 for Series 200/300 computers).
- Every character in a string is represented by one byte which contains the character's ASCII code. If the length of the string is odd, a pad character is

appended to the string to get an even number of characters; however, the length header does not include this pad character.

Examples

If stored as a string value, the number “45” would be:

$$\underbrace{00000000\ 00000000\ 00000000\ 00000010}_{\text{Length = 0002 (binary)}}\ \underbrace{00110100}_{\text{ACSII 52}}\ \underbrace{00110101}_{\text{ASCII 53}}$$

The string “A” would be stored:

$$\underbrace{00000000\ 00000000\ 00000000\ 00000001}_{\text{Length = 0001 (binary)}}\ \underbrace{01000001}_{\text{ACSII 65}}\ \underbrace{00100000}_{\text{ASCII 32}}$$

In this case, the space character (ASCII code 32) is used as the pad character; however, not all operations use the space as the pad character.

ASCII and Custom Data Representations

When using the ASCII data format for BDAT files, all data items are represented with ASCII characters. With user-defined formats, the image specifiers referenced by the OUTPUT or ENTER statement are used to determine the data representation (which is ASCII characters).

```
OUTPUT @File USING "SDD.DD,XX,B,#";Number,Binary_value
ENTER  @File USING "B,B,40A,%";Bin_val1,Bin_val2,String$
```

Using both of these formats with BDAT files produce results identical to using them with devices. For further information about OUTPUT, ENTER, and TRANSFER operations, refer to the *HP BASIC Programming Guide*.

Data Representations with HP-UX Files

HP-UX files are *very similar to BDAT files*. The *only differences* between the two are:

- The internal representation (FORMAT OFF) of strings is slightly different:
  - □ HP-UX FORMAT OFF strings have no length header; instead, they are terminated by a null character, CHR$(0).
  - □ BDAT FORMAT OFF strings have a 4-byte length header;

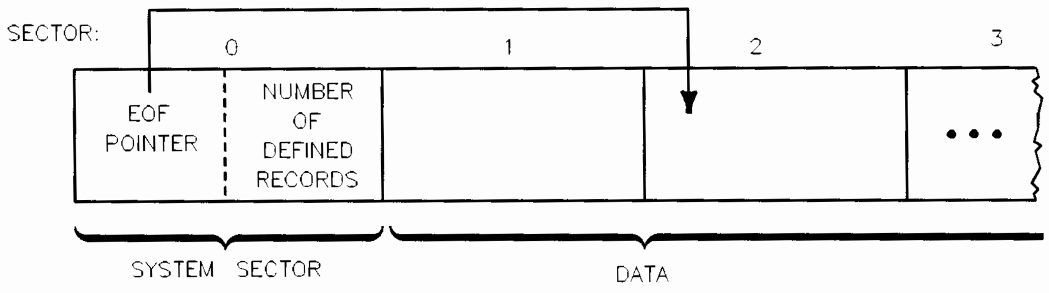
- HP-UX files have a *fixed record length of 1*. (BDAT files allow user-definable record lengths.)
- HP-UX files have *no system sector* like BDAT files do (see the next section for details).

The FORMAT ON representations for HP-UX files are the same as for devices.

Note Throughout this section, you should be able to assume that—unless otherwise stated—the techniques shown will apply to both BDAT and HP-UX files.

BDAT File System Sector

On the disk, every BDAT file is preceded by a system sector that contains an end-of-file (EOF) pointer and the number of defined records in the file. All data is placed in succeeding sectors. You cannot directly access the system sector. However, as you shall see later, it is possible to indirectly change the value of an EOF pointer.



EOF Pointer:	• number of sectors from beginning of file (32-bit binary number)
	• number of bytes from beginning of sector (32-bit binary number)
Number of defined records:	See description below (32-bit binary number)

Defined Records

To access a BDAT or HP-UX file randomly, you specify a particular defined record. Records are the smallest units in a file directly addressable by a random OUTPUT or ENTER.

- With BDAT files, defined records can be anywhere from 1 through 65 534 bytes long.
- With HP-UX files, defined records are always 1 byte long.

Specifying Record Size (BDAT Files Only)

Both the length of the file and the length of the defined records in it are specified when you create a BDAT file. This section shows how to specify the record length of a BDAT file. (The next section talks about how to choose the record length.)

For example, the following statement would create a file called `Example` with 7 defined records, each record being 128 bytes long:

```
CREATE BDAT "Example",7,128
```

If you don't specify a record length in the CREATE BDAT statement, the system will set each record to the default length of 256 bytes.

The number of records is rounded to the nearest integer (odd or even). The record length must be an even number of bytes, so it is rounded to the nearest integer, and then is rounded again to the nearest *even* integer as shown in the following table:

How Record Lengths are Rounded

Record Length in Statement	Final Record Length
record length < 0.5	Error
$0.5 \leq$ record length < 2.5	2
$2.5 \leq$ record length < 4.5	4
$4.5 \leq$ record length < 6.5	6
⋮	⋮
and so forth	

For example, the statement:

```
CREATE BDAT "Odd",3.5,28.7
```

would create a file with 4 records, each 30 bytes long. On the other hand, the statement:

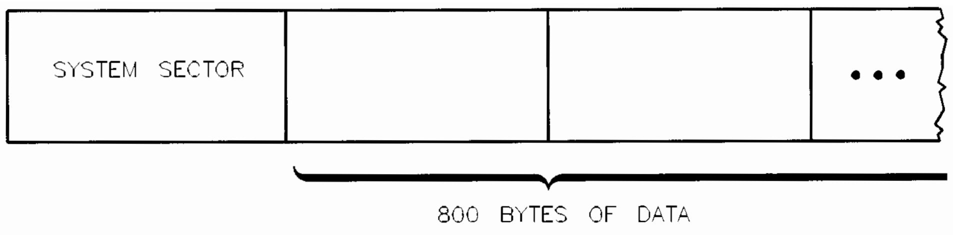
```
CREATE "Odder",3.49,28.3
```

would create a file with 3 records, each 28 bytes long.

Once a file is created, you cannot change its length, or the length of its records. You must therefore calculate the record size and file size required *before* you create a file.

Choosing a Record Length (BDAT Files Only)

Record length is important only for random OUTPUTs and ENTERs. It is not important for serial access. The most important consideration in selecting of a proper record length is the type of data being stored and the way you want to retrieve it. Suppose, for instance, that you want to store 100 real numbers in a file, and be able to access each number individually. Since each REAL number uses 8 bytes, the data itself will take up 800 bytes of storage.



The question is how to divide this data into records. If you define the record length to be 8 bytes, then each REAL number will fill a record. To access the 15th number, you would specify the 15th record. If the data is organized so that you are always accessing two data items at a time, you would want to set the record length to 16 bytes.

The worst thing you can do with data of this type is to define a record length that is not evenly divisible by eight. If, for example, you set the record length to four, you would only be able to randomly access half of each real number at a time. In fact, the system will return an End-Of-Record condition if you try to randomly read data into REAL variables from records that are less than 8 bytes long.

So far, we have been talking about a file that contains only REAL numbers. For files that contain only INTEGERs, you would want to define the record length to be a multiple of two. To access each INTEGER individually, you would use a record length of two; to access two INTEGERs at a time, you would use a record length of four, and so on.

Files that contain string data present a slightly more difficult situation since strings can be of variable length. If you have three strings in a row that are 5, 12, and 18 bytes long, respectively, there is no record length less than 22 that will permit you to randomly access each string. If you select a record length of 10, for instance, you will be able to randomly access the first string but not the second and third.

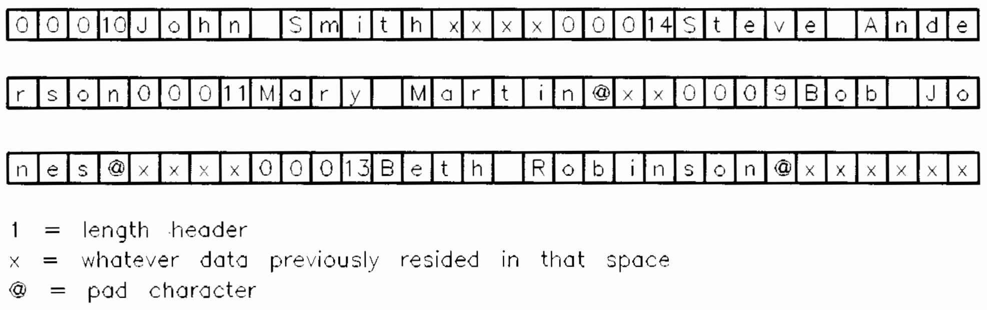
If you want to access strings randomly, therefore, you should make your records long enough to hold the largest string. Once you've done this, there are two ways to write string data to a BDAT file. The first, and easiest, is to output each string in random mode. In other words, select a record length that will hold the longest string and then write each string into its own record. Suppose, for example, that you wanted to OUTPUT the following 5 names into a BDAT file and be able to access each one individually by specifying a record number.

```
John Smith
Steve Anderson
Mary Martin
Bob Jones
Beth Robinson
```

The longest name, "Steve Anderson", is 14 characters. To store it in a BDAT file would require 18 bytes (four bytes for the length header). So you could create a file with record length of 18 and then OUTPUT each item into a different record:

```
100   CREATE BDAT "Names",5,18          ! Create a file.
110   ASSIGN @File TO "Names"           ! Open the file (FORMAT OFF).
120   OUTPUT @File,1;"John Smith"       ! Write names to
130   OUTPUT @File,2;"Steve Anderson"   !  successive records
140   OUTPUT @File,3;"Mary Martin"      !  in file.
150   OUTPUT @File,4;"Bob Jones"
160   OUTPUT @File,5;"Beth Robinson"
```

On the disk, the file `Names` would look like the figure below. The four-byte length headers show the decimal value of the bytes in the header. The data are shown in ASCII characters.


0 0 0 10 J o h n   S m i t h x x x x 0 0 0 14 S t e v e   A n d e
r s o n 0 0 0 11 M a r y   M a r t i n @ x x 0 0 0 9 B o b   J o
n e s @ x x x x 0 0 0 13 B e t h   R o b i n s o n @ x x x x x x
1 = length header
x = whatever data previously resided in that space
@ = pad character

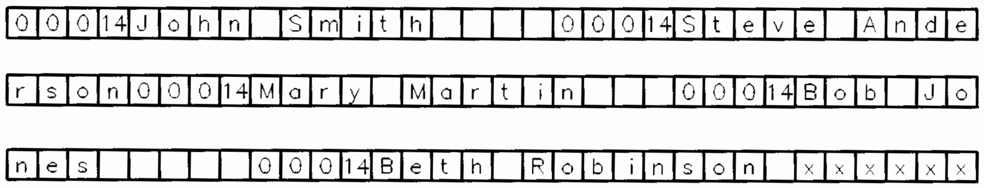
The unused portions of each record contain whatever data previously occupied that physical space on the disk.

The other method for writing strings to a BDAT or HP-UX file is to pad each entry so that they are all of uniform length. While this method involves more programming, it allows you to pad the unused portions of each record with whatever characters you choose. It also permits you to read and write the data serially as well as randomly. The program below shows how you might enter the five names into a file by padding each name with spaces.

```
100   CREATE BDAT "Names",5,18     ! Create file.
110   ASSIGN @Path1 TO "Names"     ! Open the file (FORMAT OFF).
120   FOR Entry=1 TO 5
130      LINPUT Name$[1;14]        ! Get names from keyboard.
140      OUTPUT @Path1;Name$       ! Write name to file.
150   NEXT Entry
```

In this program, we input each name from the keyboard and then pad the name with spaces so that its length is 14 bytes. With the four-byte length header, each entry is 18 bytes, or one record. In line 140, we write the name serially to the file. Since every data item is 18 bytes, there is no need to write randomly, although we could have if we wanted to. Since the LINPUT statement is limited to 14 bytes, any names that are longer than 14 characters are automatically truncated.

If we had used the second program to enter the names, file **Names** would look like the figure below:


0 0 0 14 J o h n S m i t h 0 0 0 14 S t e v e A n d e
r s o n 0 0 0 14 M a r y M a r t i n 0 0 0 14 B o b J o
n e s 0 0 0 14 B e t h R o b i n s o n x x x x x x

EOF Pointers

There are two types of End-Of-File pointers associated with BDAT and HP-UX files:

- Logical EOF pointer in the I/O path table—maintained in the table of the I/O path currently assigned to the file.
- Logical EOF pointer on the volume—resides on the physical volume that contains the file:
  - □ With BDAT files, it is in the system sector.
  - □ With HP-UX files, the EOF pointer is stored in one of two places:

- On HFS volumes, it is read from the size stored in the file's inode.
- On LIF volumes, it is read from a long word stored in the directory.

The two pointers are always updated at the same time so that they always agree with one another. (This may not be true if you use more than one I/O path to OUTPUT data to one file.) The two pointers are updated when either of the two conditions below occur.

- If, after an OUTPUT statement has been executed, the file pointer value is greater than the EOF pointers, the EOF pointers are moved to the file pointer position.
- If an OUTPUT statement contains the "END" secondary word, the EOF pointers are moved to the file pointer position regardless of their current values.

The function of EOF pointers is to mark the logical end of a data file. Every file also has a physical EOF on a volume—the last byte reserved for the file when you create it. The EOF pointers cannot point beyond the physical EOF. The EOF pointer marks the point at which no more data can be read. Also, you cannot randomly write data more than one record past the current EOF position.

If you have a 100-record file, and the EOF pointers point to the 50th record, records 50 through 100 cannot be read. If you attempt to read data beyond an EOF pointer, an EOF condition occurs. EOF conditions can be trapped with an ON END statement. If you do not trap it, an EOF condition will cause Error 59. Attempting to read or write beyond the physical EOF on a volume will also result in an EOF condition. EOF conditions are described in more detail later in this chapter. Note that files on SRM and HFS disks are extensible. Thus the file is extended rather than getting a physical EOF condition.

Moving EOF Pointers

When you first create a file, the logical EOF on a volume has a pointer which points to the first byte in the file. When you ASSIGN an I/O path to a file, the logical EOF pointer on a volume is copied to the I/O path table. As you OUTPUT data items to the file, both EOF pointers are moved so that they point to the next byte. This is also where the file pointer is positioned.

If you overwrite a file, however, the EOF pointers will not necessarily agree with the file pointer. For example, suppose you write 100 bytes to a file, and then re-ASSIGN the I/O path. By re-ASSIGNing, you move the file pointer back to the first byte in the file. The EOF pointers, though, still point to the 101st byte. They will not be changed until the file pointer value is greater than 101, or until you specify an "END" in an OUTPUT statement. The EOF pointers can also be set using CONTROL registers 7 and 8.

The secondary word "END" is used to move the EOF pointers backwards. It forces the EOF pointers to be re-positioned to the file pointer byte even if it is earlier in the file than their current position. In effect, this shrinks the file, causing data that lies past the new EOF position to become inaccessible.

Writing Data

Data is always written to a file with an OUTPUT statement via an I/O path. You can OUTPUT numeric and string variables, numeric and string expressions, and numeric and string arrays. When you OUTPUT data with the FORMAT OFF, data items are written to the file in internal format (described earlier).

There is no limit to the number of data items you can write in a single OUTPUT statement, except that program statements are limited to two CRT lines. Also, if you try to OUTPUT more data than the file can hold, or the record can hold (if you are using random access), the system will return an EOF or EOR condition. If an EOF or EOR condition occurs, the file retains any data output before the end condition occurred.

There is also no restriction on mixing different types of data in a single OUTPUT statement. The system decides which data type each item is before it writes the item to the disk. Any item enclosed in quotes is a string. Numeric variables and expressions are OUTPUT according to their type (16 bytes for COMPLEX values, 8 bytes for REAL values, and 2 bytes for INTEGER values). Arrays are written to the file in row-major order (right-most subscript varies quickest).

Each data item in an OUTPUT statement should be separated by either a comma or semi-colon (there is no operational difference between the two separators with FORMAT OFF). Punctuation at the end of an OUTPUT statement is ignored with FORMAT OFF.

Serial OUTPUT

Data is written serially to BDAT and HP-UX files whenever you do not specify a record number in an OUTPUT statement. When writing data serially, each data item is stored immediately after the previous item (with FORMAT OFF in effect, there are no separators between items). Sector and record boundaries are ignored. Data items are written to the file one by one, starting at the current position of the file pointer. As each item is written, the file pointer is moved to the byte following the last byte of the preceding item. After all of the data items have been OUTPUT, the file pointer points to the byte following the last byte just written.

There are a number of circumstances where it is faster and easier to use serial access instead of random access. The most obvious case is when you want to access the entire file sequentially. If, for example, you have a list of data items that you want to store in a file and you know that you will never want to read any of the items individually, you should write the data serially. The fastest way to write data serially is to place the data in an array and then OUTPUT the entire array at once.

Another situation where you might want to use serial access is if the file is so small that it can fit entirely into internal memory at once. In this case, even if you want to change individual items, it might be easier to treat the entire file as one or more arrays, manipulate as desired, and then write the entire array(s) back to the file.

Note Most of the details of the subsequent examples also pertain to how data items are written to HP-UX files. The main difference is that HP-UX files always have a defined record length of 1 byte.

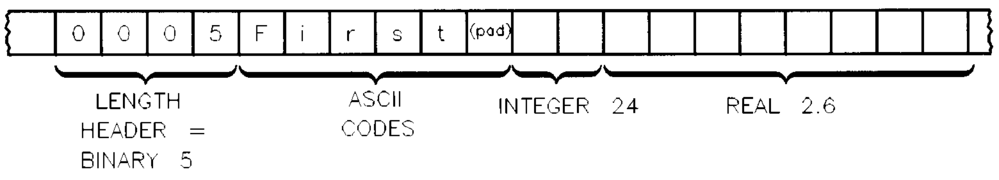
The examples below illustrate how data is stored serially in a BDAT file. Assume that the following statement was used to open the file (and assign the FORMAT OFF attribute to the I/O path):

```
ASSIGN @Path1 TO "BDATorHPUX"; FORMAT OFF
```

The statement:

```
OUTPUT @Path1;"First",24;2.6,
```

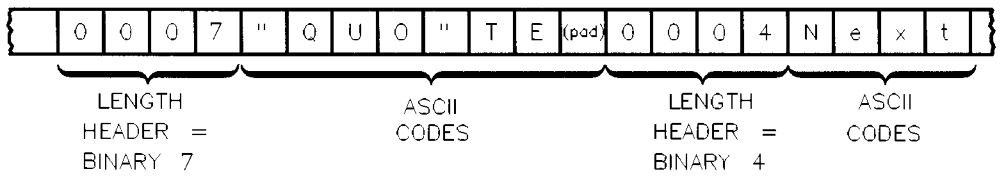
would result in the following storage format:



Note that quotation marks around a string are not written to the file. To write quote marks to a file, enter two quote marks for every one you want to OUTPUT. Note also that separators are not written to the file. To write a comma or semi-colon to a file, you must enclose it in quotes. For instance, the statement:

```
OUTPUT @Path1; """QUO""TE","Next"
```

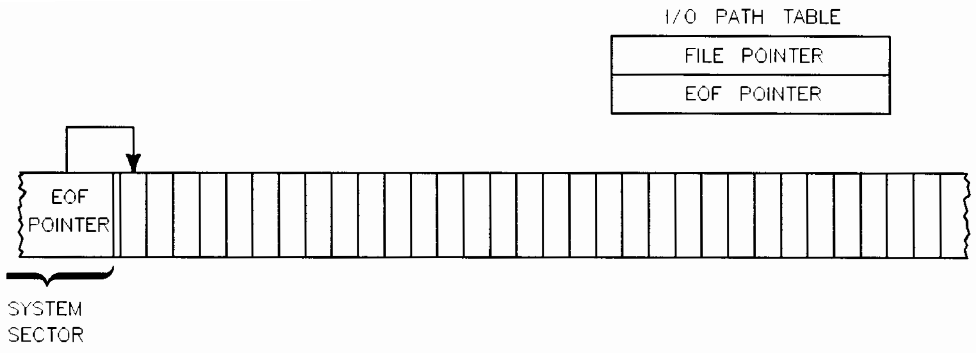
would be stored:



The following sequence of serial OUTPUT statements show how data is written to a BDAT file and how the file pointer and EOF pointers are updated.

The following statement creates a BDAT file with four 128-byte records.

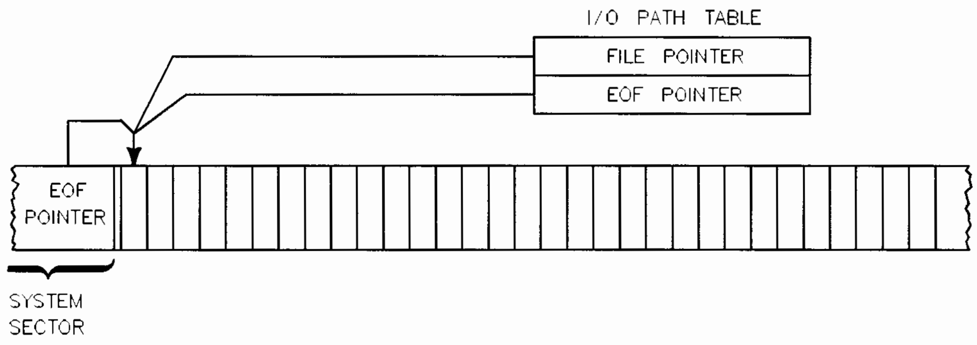
```
CREATE BDAT "Example",4,128
```



When the file is initially created, the logical EOF pointer on the volume points to the first byte in the file.

The following statement opens an I/O path to the file named **Example**.

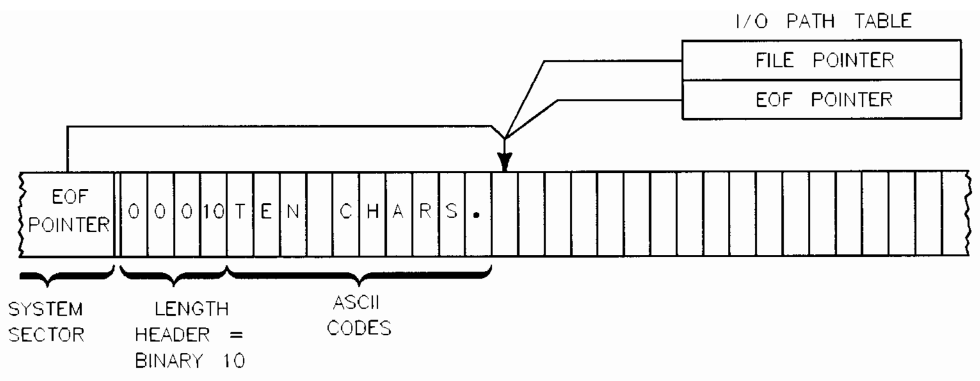
```
ASSIGN @Path1 TO "Example"
```



The logical EOF pointer on the volume is copied from the volume into the I/O path table. The file pointer is positioned to the beginning of the file.

Fourteen bytes are written to the file with this statement.

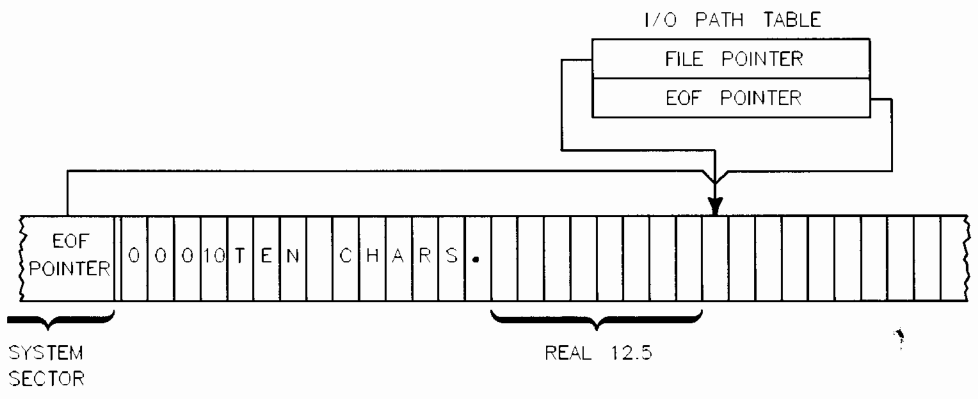
```
OUTPUT @Path1;"TEN CHARS."
```



The EOF pointers are moved to the 15th byte. The file pointer also points to the 15th byte.

This statement writes eight more bytes to the file.

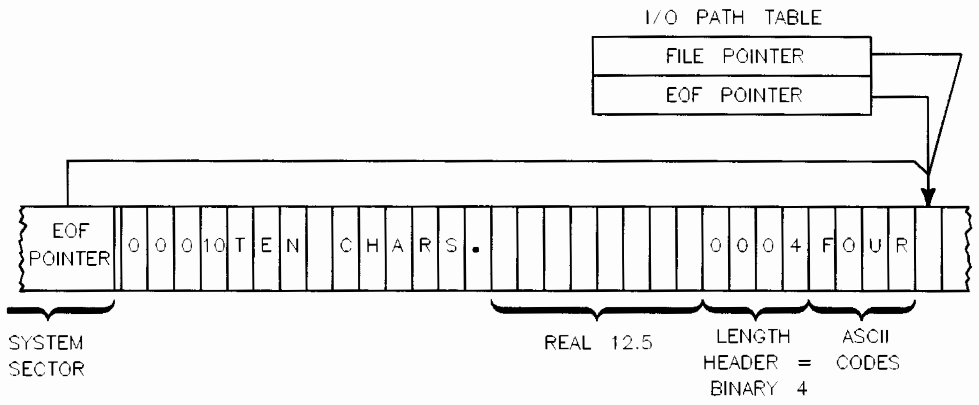
```
OUTPUT @Path1;12.5,END
```



The file pointer now points to the 23rd byte. Both the logical EOF pointer in the I/O path table and the logical EOF pointer on the volume are updated to 23.

This statement writes eight more bytes to the file.

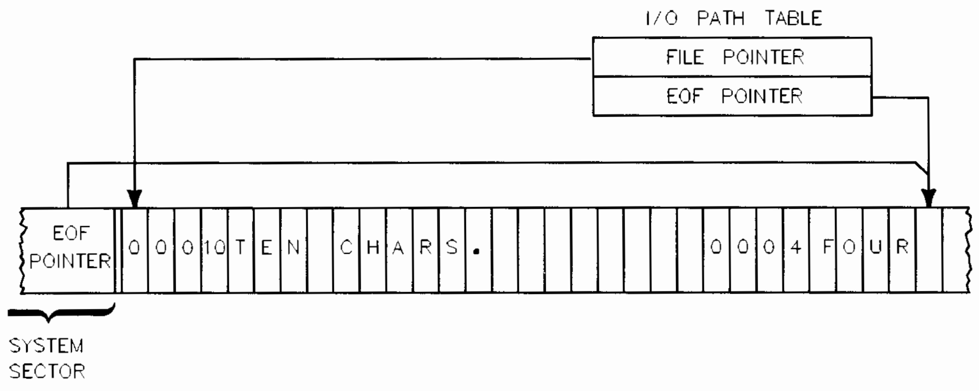
```
OUTPUT @Path1;"FOUR"
```



The file pointer now points to the 31st byte. The EOF pointers are updated to 31 because 31 is greater than 23, the current EOF value.

This statement re-assigns the I/O path name, which re-opens the file.

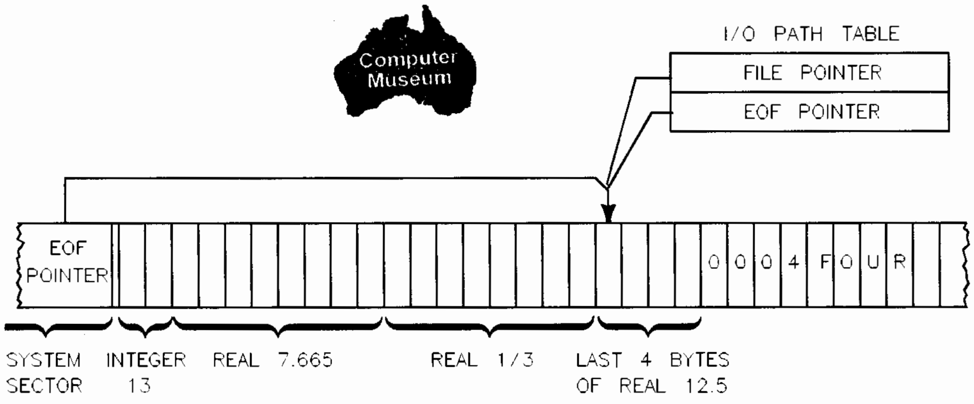
```
ASSIGN @Path1 TO "Example"
```



The file pointer is positioned back to the beginning of the file. The value of logical EOF pointer on a volume is copied into the logical EOF pointer in the I/O path table.

In this example, eighteen bytes (one INTEGER and two REALs) are OUTPUT, starting at the beginning of the file.

```
OUTPUT @Path1;13,7.665,1/3,END
```



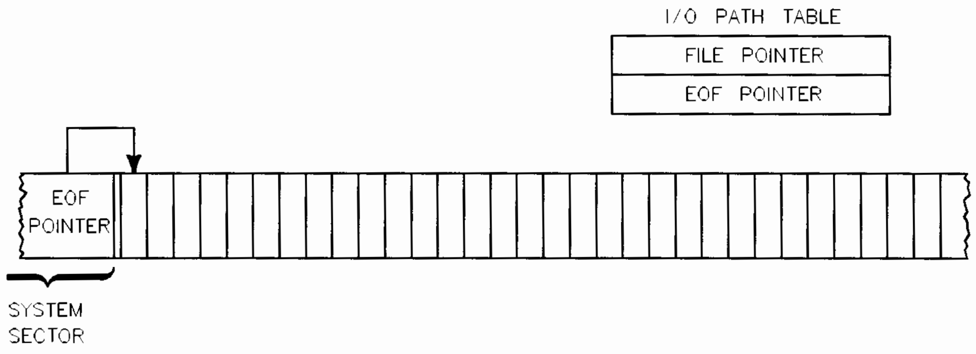
The original data, therefore, is overwritten. The file pointer points to the 19th byte. The EOF pointers are also positioned to 19 because the statement contains the "END" secondary word.

Random OUTPUT

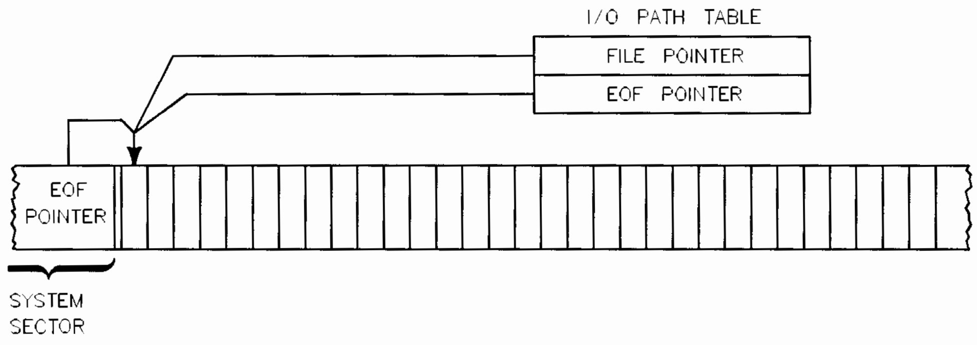
Random OUTPUT allows you to write to one record at a time. As with serial OUTPUT, there are EOF and file pointers that are updated after every OUTPUT. The EOF pointers follow the same rules as in serial access. The file pointer positioning is also the same, except that it is moved to the beginning of the specified record before the data is OUTPUT. If you wish to write randomly to a newly created file, either use a CONTROL statement to position the EOF in the last record, or start at the beginning of the file and write some "dummy" data into every record.

If you attempt to write more data to a record than the record will hold, the system will report an End-Of-Record (EOR) condition. An EOF condition will result if you try to write data more than one record past the EOF position. EOR conditions are treated by the system just like EOF conditions, except that they return Error 60 instead of 59 if they are not trapped by ON END. Data already written to the file before an EOR condition arises will remain intact. The examples below illustrate how data is stored randomly in a BDAT file. (HP-UX files can also be accessed randomly; however, you may recall that HP-UX files always have record lengths of 1 byte. Examples of accessing 1-byte records are shown in the subsequent section.)

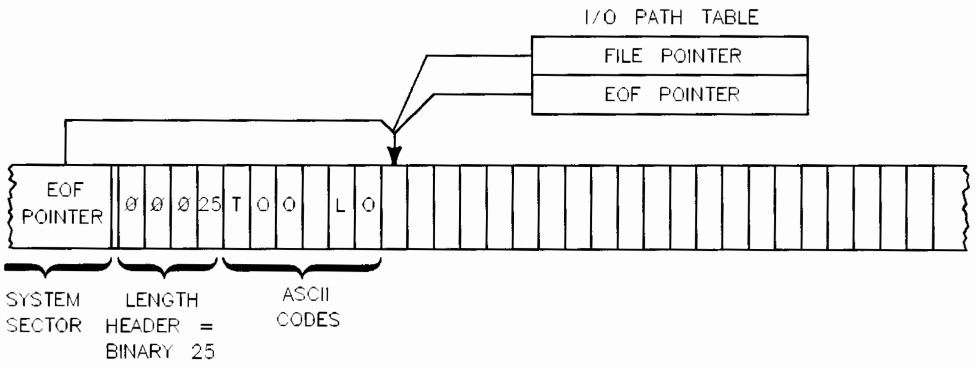
```
CREATE BDAT "Random",10,10
```



```
ASSIGN @Path2 TO "Random"
```

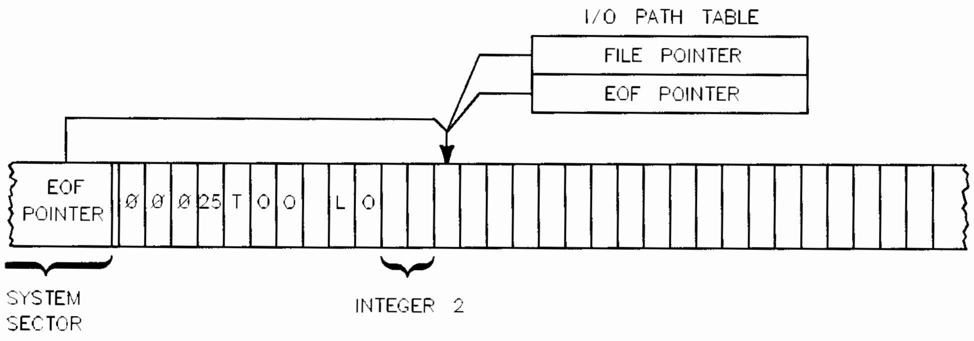


```
OUTPUT @Path2,1;"TOO LONG TO FIT IN RECORD"
```

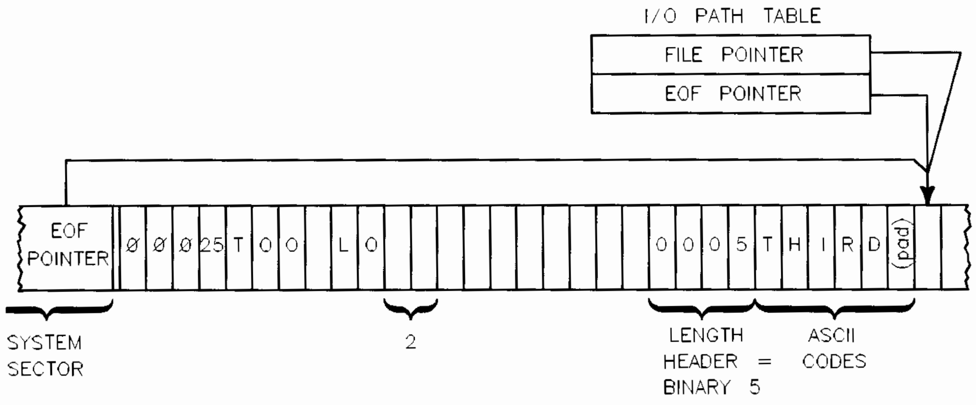


Even though this statement produces an EOR condition, the EOF pointers and file pointer are still updated. The ON END statement can be used to trap the error. Also, the length header represents the length of the string characters sent to the file, even though the whole string is not written out.

```
OUTPUT @Path2,2;2
```

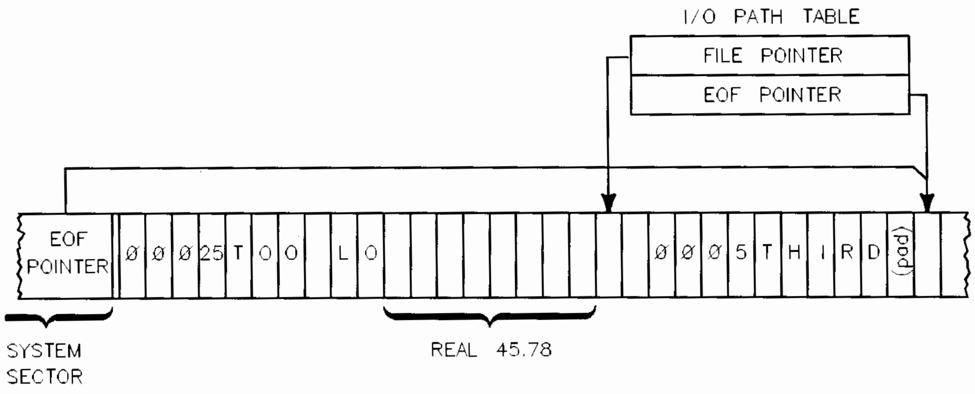


```
OUTPUT @Path2,3;"THIRD"
```



4

```
OUTPUT @Path1,2;45.78
```



Reading Data From BDAT and HP-UX Files

Data is read from files with the ENTER statement. As with OUTPUT, data is passed along an I/O path. You can use the same I/O path you used to OUTPUT the data or you can use a different I/O path.

You can have several variables in a single ENTER statement. Each variable must be separated from the other variables by either a comma or semi-colon.

It is extremely important to make sure that your variable types agree with the data types in the file. If you wrote a REAL number to a file, you should ENTER it into a REAL variable; INTEGERs should be entered into INTEGER variables; and strings into string variables. The rule to remember is:

Read it the way you wrote it.

That is the *only* technique that is always guaranteed to work.

In addition to making sure that data types agree, it is also advisable to make sure that access modes agree. If you wrote data serially, you should read it serially; and if you wrote it randomly, you should read it randomly. There are a few exceptions to this rule which we discuss later. However, you should be aware that mixing access modes can lead to erroneous results unless you are aware of the precise mechanics of the file system.

Reading String Data From a File

When reading string data from a file, you must enter it into a string variable. How the system does this depends on the file type and FORMAT attribute assigned to the file:

- With FORMAT OFF assigned to a BDAT file, BASIC reads and interprets the first four bytes after the file pointer as a length header. It will then try to ENTER as many characters as the length header indicates. If the string has been padded by the system to make its length even, the pad character is not read into the variable.
- With FORMAT OFF assigned to an HP-UX file, strings have no length header. Instead, they are assumed to be null-terminated; that is, entry into the string terminates when a null character, CHR$(0), is encountered.
- With FORMAT ON assigned to either type of file, the system reads and interprets the bytes as ASCII characters. The rules for item and ENTER-statement termination match those for devices (see the "Entering Data" chapter of the *HP BASIC Programming Guide* for details.)

After an ENTER statement has been executed, the file pointer is positioned to the next unread byte. If the last data item was a padded string (written to a BDAT file when using FORMAT OFF), the file pointer is positioned after the pad. If you use the same I/O path name to read and write data to a file, the file pointer will be updated after every ENTER and OUTPUT statement. If you use different I/O path names, each will have its own file pointer which is independent of the other. However, be aware that each also has its own EOF pointer and that these pointers may not match, which can cause problems.

Entering data does not affect the EOF pointers. If you attempt to read past an EOF pointer, the system will report an EOF condition.

Serial ENTER

When you read data serially, BASIC enters data into variables starting at the current position of the file pointer and proceeds, byte by byte, until all of the variables in the ENTER statement have been filled. If there is not enough data in the file to fill all of the variables, the system returns an EOF condition. All variables that have already taken values before the condition occurs retain their values.

The following program creates a BDAT file, assigns an I/O path name to the file (with default FORMAT OFF attribute), writes five data items serially, and then retrieves the data items.

```
10   CREATE BDAT "STORAGE",1    ! Could also be an HP-UX file.
20   ASSIGN @Path TO "STORAGE"
30   INTEGER Num,First,Fourth
40   Num=5
60   OUTPUT @Path;Num,"squared"," equals",Num*Num,"."
70   ASSIGN @Path TO "STORAGE"
80   ENTER @Path;First,Second$,Third$,Fourth,Fifth$
90   PRINT First;Second$;Third$,Fourth,Fifth$
100  END
```

```
5 squared equals 25.
```

Note that we re-ASSIGNed the I/O path in line 70. This was done to re-position the file pointer to the beginning of the file. If we had omitted this statement, the ENTER would have produced an EOF condition.

Random ENTER

When you ENTER data in random mode, the system starts reading data at the beginning of the specified record and continues reading until either all of the variables are filled or the system reaches the EOR or EOF. If the system comes to the end of the record before it has filled all of the variables, an EOR condition is returned, which can be trapped by ON END.

In the following example, we randomly OUTPUT data to 5 successive records, and then ENTER the data into an array in reverse order.

```
10      CREATE BDAT "SQ_ROOTS",5,2*8
20      ASSIGN @Path TO "SQ_ROOTS"  ! Default is FORMAT OFF.
30      FOR Inc=1 to 5
40         OUTPUT @Path,Inc;Inc,SQR(Inc) ! Outputs two 8-byte REALs each time.
50      NEXT Inc
60      FOR Inc=5 TO 1 STEP -1
70         ENTER @Path,Inc;Num(Inc),Sqroot(Inc)
80      NEXT Inc
90      PRINT "Number","Square Root"
100     FOR Inc=1 TO 5
110        PRINT Num(Inc),Sqroot(Inc)
120     NEXT Inc
130     END
```

```
Number     Square Root
 1          1
 2          1.41421356237
 3          1.73205080757
 4          2
 5          2.2360679775
```

In this example, there was no need to re-ASSIGN the I/O path because the random ENTER automatically re-positions the file pointer.

Line 40 of the above program outputs two 8-byte REALs to the BDAT file called SQ_ROOTS. Note that this line would have to be changed for outputs made to HP-UX files because HP-UX files always have a record length of one. For example, the OUTPUT statement would look like this:

```
OUTPUT @Path,((Inc-1)*2*8)+1;Inc,SQR(Inc)
```

And the ENTER statement would look like this:

```
ENTER @Path,((Inc-1)*2*8)+1;Num(Inc),Sqroot(Inc)
```

Executing a random ENTER without a variable list has the effect of moving the file pointer to the beginning of the specified record. This is useful if you want to serially access some data in the middle of a file. Suppose, for instance, that you have a BDAT file containing 100 8-byte records, and each record has a REAL number in it. If you want to read the last 50 data items, you can position the file pointer to the 51st record and then serially read the remainder of the file into an array.

```
100    REAL Array(50)
110    ENTER @Realpath,51;     ! 51*8 is HP-UX record number.
120    ENTER @Realpath;Array(*)
```

Accessing Files with Single-Byte Records

With BDAT files, you can define records to be just one byte long (defined records in HP-UX files are always 1 byte long). In this case, it doesn't make sense to read or write one record at a time since even the shortest data type requires two bytes to store a number.

Random access to one-byte records, therefore, has its own set of rules. When you access a one-byte record, the file pointer is positioned to the specified byte. From there, the access proceeds in serial mode. Random OUTPUTs write as many bytes as the data item requires, and random ENTERs read enough bytes to fill the variable.

The example below illustrates how you can read and write randomly to one-byte records.

```
10     INTEGER Int
20     CREATE BDAT "BYTE",100,1
30     ASSIGN @Bytepath TO "BYTE"
40     OUTPUT @Bytepath,1;3.67
50     OUTPUT @Bytepath,9;3
60     OUTPUT @Bytepath,11;"string"
70     ENTER @Bytepath,9;Int
80     ENTER @Bytepath,1;Real
90     ENTER @Bytepath,11;Str$
100    PRINT Real
110    PRINT Int
120    PRINT Str$
130    END
```

```
 3.67
 3
 string
```

Note that we had to declare the variable `Int` as an INTEGER. If we hadn't, the system would have given it the default type of REAL and would therefore have required 8 bytes.

Trapping EOF and EOR Conditions

An EOF condition exists whenever the system attempts to read data at, or beyond, the byte marked by the EOF pointers. The EOR condition will arise if you attempt to randomly read or write beyond the particular record specified. If, for example, you try to randomly OUTPUT a 20-character string into a 10-byte record, an EOR condition will occur. EOF conditions will also result whenever you try to read or write beyond the physical end-of-file.

EOF and EOR conditions can be trapped with an ON END statement. ON END is similar to ON ERROR except that it only traps EOF/EOR conditions and is only applicable to the specified I/O path. If you do not have an ON END statement in a program, the EOF/EOR condition will produce an error that is trappable by the ON ERROR statement. Encountering a logical or physical end of file will produce Error 59. Encountering an end of record in random mode produces Error 60.

You can have any number of ON END statements in a program context. ON END statements that refer to different I/O paths will not interfere with each other, even if the paths go to the same file. If you have more than one ON END to the same I/O path, the system will use whichever one it most recently executed during program flow.

An ON END is canceled by the OFF END statement. OFF END only cancels the ON END branch for the specified I/O path. Re-ASSIGNing an I/O path will also cancel any existing ON END branch for the particular path.

5

Color and Gray-Scale Graphics

Color can be used for emphasis, for clarity, and just to present visually pleasing images. Color used carefully is a valuable communication tool. Misused, color can garble communication.

The biggest benefit of the color computer is that it makes experimenting with color so easy. Also, it is possible to create simple animation effects and impressive images.

The *Manual Examples* disk (`/usr/lib/rmb/demo/manex` for BASIC/UX users) contains programs found in this chapter. As you read through the following sections, load the appropriate program and run it. Experiment with the programs until you are familiar with the demonstrated concepts and techniques.

Non-Color-Mapped Color

Two types of color displays are available with Series 200/300/400/700 computers: color-mapped displays and non-color-mapped displays. This section discusses the latter type. (Subsequent sections discuss color-mapped displays.)

Note Color-mapped displays can be used in non-color-mapped mode, as described in this section. However, the color-mapped mode is the *only* mode supported by BASIC/UX in the X Windows environment. Refer to the "Customizing X Window Colors for HP BASIC/UX" section in the "Installing HP BASIC/UX on to HP-UX" chapter of the *Installing and Maintaining HP BASIC/UX* manual for X Windows colormap details.

Specifying a Non-Color-Mapped Display

When either of the following statements is executed:

```
PLOTTER IS CRT,"INTERNAL"
```

or

```
PLOTTER IS 28,"98627A"
```

a non-color-mapped display is selected as the plotting device. The first statement chooses the non-color-mapped mode, even though the display may be capable of operating in color-mapped mode.

Available Colors

With non-color-mapped displays, eight colors are available using PEN and AREA PEN:

- black and white
- red, green, and blue (the additive color primaries)
- cyan, magenta, and yellow (the complements of the additive color primaries)

Using the HP 98627A Color Interface and Display

The HP 98627A interface allows you to connect an RGB color monitor to your computer. The HP 98627A does not support color map operations; thus, you cannot change the color of an area on the screen without redrawing that area. Nor can you define your own color-addition scheme as you can with a color-mapped device. In addition, only eight pure colors can be created on an external color monitor because there is no control over the intensity of each color gun. Each color (red, green, blue) can be either off or on—two states, three colors: $2^3=8$. To get other colors, you must dither.

If you have an HP 98627A interface connected to a 60 Hz, non-interlaced color monitor, you could send graphics to it by executing the following statement:

```
PLOTTER IS 28,"98627A"
```

Depending on your color monitor, more specification may be necessary in the string expression of the PLOTTER IS statement. (See the subsequent table for further information.)

You can connect many types of color monitors to your computer through an HP 98627A color monitor interface. In the PLOTTER IS statement, you must specify accordingly:

HP 98627A Display Formats

Desired Display Format	Plotter Specifier
Standard Graphics 512 by 390 pixels, 60 Hz, non-interlaced	`"98627A"` or `"98627A;US STD"`
512 by 390 pixels, 50 Hz, non-interlaced	`"98627A;EURO STD"`
High-Resolution Graphics 512 by 512 pixels, 46.5 Hz, non-interlaced	`"98627A;HI RES"`
TV Compatible Graphics 512 by 474 pixels, 60 Hz, interlaced (30 Hz refresh rate)	`"98627A;US TV"`
512 by 512 pixels, 50 Hz, interlaced (25 Hz refresh rate)	`"98627A;EURO TV"`

The HP98627A's display memory is composed of three "color planes" with as many bits as necessary to compose a full picture. Following is a description of how the various pen selectors affect the operation of an external color monitor.

Pen selectors are mapped into the range −7 through 7. Thus:

If pen selector>0 then use PEN (pen selector-1) MOD 7+1
If pen selector=0 then use PEN 0 (complement)
If pen selector<0 then use PEN -((ABS(pen selector)-1) MOD 7+1)

Complement means to change the state of pixels; that is, to draw lines where there are none, and to erase where lines already exist.

Non-Color-Mapped Dominant Pens

The meanings of the different pen values are shown in the table below. The pen value can draw (1), erase (0), not change (n/c), or complement (invert) the value in each memory plane.

Non-Color-Map Dominant-Pen Mode

Pen	Action	Plane 1 (red)	Plane 2 (green)	Plane 3 (blue)
-7	Erase Magenta	0	n/c	0
-6	Erase Blue	n/c	n/c	0
-5	Erase Cyan	n/c	0	0
-4	Erase Green	n/c	0	n/c
-3	Erase Yellow	0	0	n/c
-2	Erase Red	0	n/c	n/c
-1	Erase White	0	0	0
0	Complement	invert	invert	invert
1	Draw White	1	1	1
2	Draw Red	1	0	0
3	Draw Yellow	1	1	0
4	Draw Green	0	1	0
5	Draw Cyan	0	1	1
6	Draw Blue	0	0	1
7	Draw Magenta	1	0	1

Choosing Pen Colors

Colors can be selected the same way they are for other display devices—with the PEN statement. If all you want is to highlight a portion of a graph or chart, this may be all the color you need. (In non-color-map mode, other graphics displays behave exactly like the HP 98627A Color Interface Card.) The colors and their pen selectors are listed in the following table.

Default Non-Color-Map Values

Pen Value	Color	Frame Buffer Entry
0	Black	0
1	White	7
2	Red	1
3	Yellow	3
4	Green	2
5	Cyan	6
6	Blue	4
7	Magenta	5

In this mode you can draw lines in the eight colors listed above. It is possible, however, to fill areas with other shades. These tones are achieved through dithering. Dithering produces different shades by combining dots of the eight colors described above. The screen is divided into 4 x 4 cells, and patterns of dots within the cells are turned on to match, as closely as possible, the color you specify. Dithered colors are defined using AREA COLOR and AREA INTENSITY. The color models available are discussed in a subsequent section titled "Color Specification." (The actual color matching process used in dithering is described under "Dithering and Color Maps.") Filling is specified by using the secondary keyword FILL in any of the following statements:

```
IPLOT        PLOT     POLYGON
RECTANGLE    RPLOT    SYMBOL
```

GSTORE Array Sizes for the HP 98627A

As mentioned above, different color monitors display different numbers of pixels. To figure the array size necessary to GSTORE an image, multiply the number of pixels in the X direction by the number of pixels in the Y direction; multiply that by the number of color planes (3); and divide by 16 (the number of bits per word). For example, say you want to calculate the array size needed to store an image created on a U.S. standard monitor: 512x390x3÷16=37 440 words. However, you cannot specify an array which has more than 32 767 elements in any dimension. To get around this restriction, make one dimension the number of memory planes (3) and the other dimension the number of pixels

(512×390÷16). Thus, the statement declaring an array for storing an image from a U.S. standard external color monitor could look like this:

```
INTEGER Image(1:12480,1:3)
```

If your array is larger than necessary to store an image, it will be filled only to the point where the image is exhausted. If your image is larger than your array, the array will be filled completely, and the remainder of the image will be ignored.

GSTORE and GLOAD store the graphics image into this array and load it back into graphics memory, respectively.

Using Color-Map Displays

If you are trying to define a complex human interface, you will need more colors and more control over the colors. The system described in the rest of this chapter (except for dithering) is available only after you turn on the color map. To do so, execute:

```
PLOTTER IS CRT,"INTERNAL";COLOR MAP
```

If you have a an HP 98782A color monitor connected through an HP 98700 display controller and it is set to address 25, you could execute:

```
PLOTTER IS 25,"INTERNAL";COLOR MAP
```

In this way, plots can easily be plotted on various devices with a minimum of programming effort.

The Frame Buffer

Most Series 200/300 and all Series 400/700 color displays have bit-mapped color graphics. An area in memory called a **frame buffer** provides 1, 4, 6, or 8 bits of memory for each pixel location. (The number of bits available for describing each pixel is sometimes called the **depth** of the frame buffer.) A 4-bit frame buffer allows each pixel location to contain a number between 0 and 15 (inclusive). Thus a four-plane frame buffer can display lines in 16 different colors on the CRT, simultaneously. At any given time, the values written to the frame buffer fall into four categories:

Background Value	Whenever GCLEAR is executed, all the pixel locations in the frame buffer are set to 0. Thus, 0 is the background color.
Line Value	The PEN statement determines the value written to the frame buffer for all lines drawn. This includes all lines (including characters created by LABEL) and outlines (specified by the secondary keyword EDGE).
Fill Value	The AREA PEN statement specifies the value written to the frame buffer for filling areas (specified by the secondary keyword FILL).
Dithered Colors	AREA INTENSITY and AREA COLOR can specify a fill color, but the results can be surprising when the COLOR MAP option has been selected (see "Dithering and Color Maps"). In addition, the dithered colors tend to introduce texturing into the areas and may not accurately reproduce the color you specify.

The PEN, AREA PEN, AREA INTENSITY, and AREA COLOR statements control what are referred to as **modal attributes**. This means that the value established by one of the statements stays in effect until it is altered by another statement. (GINIT alters all of them.)

Erasing Colors

Erasing is fairly simple in frame buffers that are a single bit deep. You just restore the background by setting the portion of the frame buffer you wish to erase to 0. The concept is more complex in frame buffers with more depth. As long as the graphics system is in the dominant writing mode (see "Non-Dominant Writing"), there are three ways of erasing:

- The easiest is GCLEAR. However, GCLEAR destroys the entire image. If you want to erase only part of the image, it is necessary to be more precise.
- If you know the pen used to write the line, you can use a negative pen selector of the same magnitude. This will erase the pen value from the frame buffer. (This works for PEN and AREA PEN.)
- If you don't know the pen used to create the image, you can overwrite the image with the background color. This can be PEN 0, or, if you are on a filled area, whatever pen the area was filled with. A fairly simple extension of

this is to use the RECTANGLE statement to implement a *local GCLEAR* to erase portions of the screen.

Default Colors

If you do not modify the color map (see the next section for how to do that) the colors selected by the PEN and AREA PEN values depend on the default color map values, which are:

Default Color Map Values

Value	Color
0	Black
1	White
2	Red
3	Yellow
4	Green
5	Cyan
6	Blue
7	Magenta
8	Black
9	Olive Green
10	Aqua
11	Royal Blue
12	Maroon
13	Brick Red
14	Orange
15	Brown

Pens 16 through the end of the color map are also defined for displays with more than four planes. You can interrogate the color map with GESCAPE for exact values.

The Primary Colors

The lower eight pens of the default color map are the same as are available without enabling the color map, but they do not write the same value into the frame buffer (see the PEN statement in the *HP BASIC Language Reference*.) The colors are:

- black and white (the extremes of no-color)
- red, green, and blue (the additive primaries)
- cyan, magenta, and yellow (the subtractive primaries)

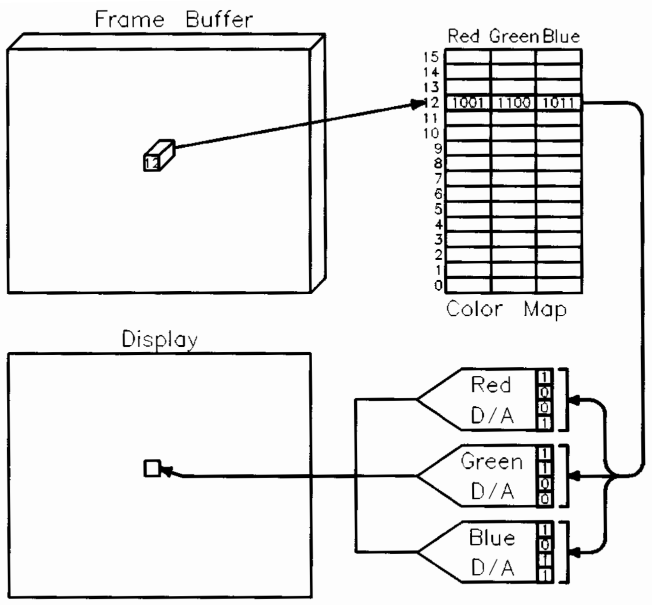
The Business Colors

The upper eight colors (8 through 15) were selected by a graphic designer to produce graphs and charts for business applications. The colors are:

- maroon, brick red, orange, and brown (warm colors)
- black, olive green, aqua, royal blue (cool colors)

The Color Map

The color-mapped system uses the value in the frame buffer as an index into a color map. The color map contains a much larger description of the color to be used and, just as importantly, the color description used is *indirect*. Thus, the value in the frame buffer does not say "use color 12," but rather "use the color described by register number 12." Note that 8-bit binary numbers are stored in the color map in the diagram below. This is not the case for a Model 236C; it stores 4-bit binary numbers. All other color-mapped systems use 8-bit binary numbers in their color map.



Color Map (Model 236C)

The CRT refresh circuitry reads the value from the pixel location in the frame buffer, uses it to look up the color value in the color map, and displays that color at that pixel location on the CRT. Thus, it is possible to draw a picture with a given set of colors in the color map (a set of colors is called a **palette**) and then change palettes and produce a new picture by redefining the colors, rather than having to redraw the picture. (The binary numbers in the color map are created by the system. The user deals with normalized values, as described under "Color Specification.")

Color Specification

The SET PEN statement is used to control the values in the color registers in the color map. The SET PEN statement supports two color models for selecting the color each pen represents, the RGB (red, green, blue) model and the HSL (hue, saturation, luminosity) model. Since the color models are dynamically interactive, it is much easier to understand them by experimenting with them.

The RGB Model (Red, Green, Blue)

The RGB model can be thought of as mixing the output of three light sources (one each of red, green, and blue). The parameters in the model specify the intensity of each of the light sources. The RGB model is accessed through the secondary keyword INTENSITY with the SET PEN statement. The RGB model is closest to the actual physical system used by color displays. The red, green, and blue values represent the value modulating the electron guns that excite the colored phosphors on the CRT. The values are *normalized* (range from 0 through 1). The normalized values are converted to 8-bit binary numbers to store in the color map. Each of the values controls an 8-bit digital-to-analog converter, providing 256 intensity levels from full-off to full-on for each of the colors. Thus,

```
SET PEN 0 INTENSITY 127/255,7/255,7/255
```

sets pen 0 (the background color) to approximately a "2.7%" gray value. (Whenever all the guns are set to the same intensity, a gray value is obtained.) It is simpler to think in 1/255ths and let the computer do the conversion to a decimal fraction, since the intensity parameters can be numeric expressions. The parameters for the INTENSITY mode of SET PEN are in the same order they appear in the name of the model (red, green, blue).

Note The Model 236C stores 4-bit binary numbers in the color map and uses 1/15ths for its intensity value.

The HSL Model (Hue, Saturation, Luminosity)

The HSL model is closer to the intuitive model of color used by artists, and is very effective for interactive color selection. The three parameters represent **hue** (the pure color to be worked with), **saturation** (the ratio of the pure color mixed with white), and **luminosity** (the brightness-per-unit area.) Run the program `NEW_MODELS` (which is on the *Manual Examples Disk*) to see a physical model of the HSL model parameters.

Hue rotates a color wheel to select a "pure" color to use. This color is then mixed with white light. The ratio of the pure colored light to the white light is controlled by the saturation slider. Finally, the output passes through the luminosity iris (think of it as a hole you can adjust the size of) that controls the brightness of the output.

The HSL model is accessed through the SET PEN statement with the secondary keyword COLOR:

```
SET PEN Current_pen COLOR H(Current_pen),S(Current_pen),L(Current_pen)
```

HSL Resolution

The resolution of the HSL model is not specified anywhere because the resolution for the various parameters is not fixed. The resolution for any parameter of the HSL system depends on all three of the parameters. The resolution is not only changed by the other two parameters, but also by the magnitude of the parameter you are varying. If resolution of the system becomes important in a program, it is possible to use a GESCAPE to read the RGB values back from the color map to watch for a change in the actual value being written in the color map. Change the HSL parameters by very small increments (on the order of 0.001) until a change in the color map entry is detected. This is best done using color map entry 0, since you will only need to read a single entry from the color map to check for the change.

Which Model?

Two models are provided for your color computer. The INTENSITY option of the SET PEN statement is faster than the COLOR option, because it directly reflects the hardware in the system. If you are working with primaries only, or want gray scale output, the RGB model is great. If, on the other hand, you are trying to use pastels and shades, you are better off with an intuitive color model, and that is where the HSL model shines.

It is possible to get the best of both worlds by using the HSL model for the human interaction, then reading the color map with a GESCAPE statement to get the RGB color values. The RGB values can then be used to rapidly load a palette into the color map. This is done by reading in the RGB color map values, calculating which corner of the color cube is farthest from the background color, setting the foreground pen and text displays to that color, and then writing the RGB array back into the color map. Even though the primary interaction is with the HSL model, the RGB is used because it is more convenient to find distances between colors in it.

Note that the models can be mixed freely. There is nothing to prevent using INTENSITY to set a gray background color and a black pen, and then using

COLOR to produce the rest of the palette. Use whatever is easiest for what you want to do.

If you are interested in pursuing the color models, the RGB model is formally referred to as a color cube and the HSL model is called the color cylinder. These models represent idealized color spaces and are described under "Color Spaces" later in this chapter.

Dithering and Color Maps

In early color systems which did not control the intensity of individual pixels, dithering became a popular method of increasing the number of shades available to the computer. By reducing the effective resolution of the system, it was possible to provide a large variety of shades.

Your color computer provides dithering for applications that require more shades than 16, 64 or 256 colors that are available at any single time with the color map on your particular color-mapped system. The AREA INTENSITY and AREA COLOR statements provide access to the dithered colors, although they will fill with a single pen if the color requested exists in the current color map.

If you are not in the color-map mode, AREA INTENSITY and AREA COLOR will produce the same results on a color computer that they produce on non-color-map devices, such as the HP 98627 Color Interface Card.

Creating A Dithered Color

The following discussion gets a little abstract, and it is not absolutely necessary to understand how dithering works to use it. It is interesting information and can be useful knowledge if dithered areas don't do what you expect.

A color vector is a directed line connecting two points in RGB color space. The dithering process tries to match a **target vector** by constructing a **solution vector** from colors in the color map. The actual dithered color to be produced will be 16 times the target vector, since 16 points in the dither area will be combined to create it.

The color matching process requires sixteen steps. First, the target vector is compared to the vectors produced by all the colors in the color map. The one which is closest to the target vector, depending on the distance between points in the RGB color space, is selected as the first component of the solution vector. The RGB color space is a three-dimensional Cartesian coordinate system.

The following process is then repeated 15 times:

1. The target vector is added to itself to produce a new target vector.
2. A trial solution vector is created for each color in the color map by adding the vector for the color map entry to the previous solution vector. The trial solution vector that is closest to the target vector is selected as the new solution vector.

At this point, the target vector is 16 times the original target vector, and the solution vector consists of a summation of color vectors from the color map that produce, at each iteration, the vector closest to the target vector.

The colors are then sorted by luminosity and filled into the following precedence matrix (the most luminous color is filled into the lowest numbered pixel):

Dithering Precedence Matrix

1	13	4	16
9	5	12	8
3	15	2	14
11	7	10	6

The dither precedence matrix is actually tied to pixel locations on the CRT. The matrix is repeated across the CRT and from the top to the bottom of the CRT. Areas to be filled are mapped against the fixed dithering pattern. All dither cells completely within an outline to be filled are turned on according to the precedence pattern. Cells which are only partially within the border are only partially enabled. If the area fill pattern calls for a pixel outside the boundary to be set, it will not be.

There are problems with dithering, especially when used with the color map:

- The dithered color selection tends to produce textures. In some cases, the textures overwhelm the shade produced.
- The dithered colors are not necessarily accurate representations of the color specified. This is especially true if the color map is loaded with a palette that is less than ideal for dithering. A 4-by-4 dither cell with one full intensity green pixel does not look the same as the same cell filled with the color map color 1/15 green.
- The dithered colors are not stable if the color map is altered. (If you change the color map after doing a fill based on an AREA COLOR or AREA INTENSITY, the fill value can change.)
- The dithering operation produces anomalies when the area to be filled is thin. If it is less than four pixels wide or high, it cannot contain the entire dither cell and the results can be surprising for colors which turn on small portions of the cell.

If You Need More Colors

If you have an application that requires more than 16 (4-plane systems), 64 (6-plane systems) or 256 (8-plane systems) colors, the first thing to do is see if you can redefine it to use 16, 64 or 256 colors. In many cases this is possible, and the higher quality of the color mapped palette is worth a little checking to see if you can use it.

If you absolutely have to get at a larger palette, then load a palette optimized for dithering (optimizing for dithering is described below) and *stick with dithering*. Don't try to mix color map redefinition and dithering—it will probably cause you a lot of grief. Especially, do not try to do interactive redefinition of the color map in a system that also does dithering.

Optimizing for Dithering

The actual color palette you require determines the optimum color map values. If you have specialized needs you can create palettes that are optimum for specific applications. For example, you could load the color map with 16 shades of red to produce an optimum palette for producing an image that only contained red objects.

Non-Dominant Writing

All the techniques described so far have dealt with dominant writing to the frame buffer. In the dominant writing mode, the pen selector is written directly to the color map, and overwrites whatever is currently in the frame buffer. In non-dominant writing, a bit-by-bit logical-OR is performed on the contents of the frame buffer and the pen selector value being written to the frame buffer. Thus, if pen 1 is written to a buffer location that has already been written to with pen 6, the buffer location will contain 7, but writing pen 2 to a buffer location that has already been written to with pen 6 will not change the contents.

Non-dominant writing can be used to create a properly defined palette of colors in the color map. Using this palette of colors, it is fairly easy to create a copy of the primary color circles. An additive palette is created in lines 490 through 540, by modeling the three least-significant bits of the frame buffer as **color planes**. Bit 0 is treated as representing red, bit 1 as representing green, and bit 2 as representing blue.

```
470   !******************** Create the Additive Palette ***
480   !
490   FOR I=0 TO 7
500     Red=BIT(I,0)
510     Green=BIT(I,1)
520     Blue=BIT(I,2)
530     SET PEN I INTENSITY Red,Green,Blue
540   NEXT I
```

The palette is created in the color map and then read into an array, using GESCAPE.

```
620   GESCAPE CRT,2;Additive(*)     ! Read additive palette
```

The subtractive palette is created in lines 750 through 840. The palette is created by converting between the RGB map created for the additive palette, above, and a CMY (cyan, magenta, and yellow) system. (The technique is described in more detail in the next section, “Color Spaces.”)

```
750   FOR I=0 TO 15                     ! Create subtractive palette
760     FOR J=1 TO 3
770       Point(1,J)=Additive(I,J)      ! Read a point from additive palette
780     NEXT J
790     MAT New_point= Unit-Point
800     !
810     ! The next line prints out PEN INTENSITY values for both palettes
820     IF I<8 THEN PRINT USING Pen_image2;White$,I,Point(1,1),Point(1,2),
Point(1,3),Black$,I,New_point(1,1),New_point(1,2),New_point(1,3)
830     SET PEN I INTENSITY New_point(*) !
840   NEXT I
```

The Surprise palette is created by reading from data statements.

```
210   !************** Create the Surprise Palette ****************
220   !
230   SET PEN 0 INTENSITY .6,.6,.6       ! Gray background
240   RESTORE Surprise                   ! Make sure you read the right data
250 Surprise:  ! DATA for surprise palette
260   DATA .9          ! Pen 1
270   DATA .2          ! Pen 2
280   DATA .5          ! Pen 3
290   DATA .7          ! Pen 4
300   DATA .1          ! Pen 5
310   DATA .8          ! Pen 6
320   DATA .3          ! Pen 7
330   !
340   FOR I=1 TO 7
350     READ Hue
360     SET PEN I COLOR Hue,1,1
370   NEXT I
380   !
390   MAT Point= (.6)                    !\
400   SET PEN 8 INTENSITY Point(*)       ! \
410   SET PEN 9 INTENSITY Point(*)       !  > Pens for labels
420   MAT Point= (0)                     ! /
430   SET PEN 10 INTENSITY Point(*)      !/
440   !
450   GESCAPE 3,2,Surprise(*)
```

The surprise palette relates to no known color system, but demonstrates an important point—*the non-dominant color map is arbitrary, and can represent any system you can dream up.* You may want to write in four shades of blue, have any overlap of two pens be yellow, any overlap of three pens be orange, and any overlap of four pens be red. The following lines set up such a color map.

```
230   DIM Yellow(1:1,1:3),Orange(1:1,1:3)
240   RESTORE Colors
250   READ Yellow(*),Orange(*)
260 Colors:DATA .87,.87,0,         1,.47,0
270            !
280   SET PEN 0 INTENSITY .6,.6,.6       ! Gray background
290   SET PEN 1 INTENSITY 0,0,.4         ! 0001 - Blue Plane 1
300   SET PEN 2 INTENSITY 0,0,.6         ! 0010 - Blue Plane 2
310   SET PEN 3 INTENSITY Yellow(*)      ! 0011
320   SET PEN 4 INTENSITY 0,0,.8         ! 0100 - Blue Plane 3
330   SET PEN 5 INTENSITY Yellow(*)      ! 0101
340   SET PEN 6 INTENSITY Yellow(*)      ! 0110
350   SET PEN 7 INTENSITY Orange(*)      ! 0111
360   SET PEN 8 INTENSITY 0,0,1          ! 1000 - Blue Plane 4
370   SET PEN 9 INTENSITY Yellow(*)      ! 1001
380   SET PEN 10 INTENSITY Yellow(*)     ! 1010
390   SET PEN 11 INTENSITY Orange(*)     ! 1011
400   SET PEN 12 INTENSITY Yellow(*)     ! 1100
410   SET PEN 13 INTENSITY Orange(*)     ! 1101
420   SET PEN 14 INTENSITY Orange(*)     ! 1110
430   SET PEN 15 INTENSITY 1,0,0         ! 1111
440   !
450   GESCAPE 3,2,Surprise(*)
```

Backgrounds

One nice feature available with non-dominant writing is backgrounds that aren't altered by your foreground. By restricting your foreground to pens 0 through 7, a background written with pen 8 will not be damaged by writing over it.

Complementary Writing

The concept of complementary writing was introduced in the "Interactive Graphics" chapter, under "Making Your Own Echoes." On a color computer, the concept of a complementary pen is extended to deal with the 4-bit or 8-bit values in the color map. With the non-dominant writing mode enabled, negative pen numbers will be exclusively-ORed with the contents of the frame buffer.

The complement occurs only for the bits which are one in the pen selector. Thus a pen selector of −6 would complement bits 1 and 2 of the frame buffer. If a 1 exists in a frame buffer location and a line is drawn over it with PEN −6, a 7 will now be in that location. Writing over the pixel with the same pen selector will return it to a 1.

Making Sure Echoes Are Visible

It is important to understand that complementing is of the frame buffer, not the color map. You are responsible for making sure that the complemented frame buffer values are visible against one another. Be careful about placing the same color in two locations on the color map that are complements of one another. If you pick one of them as an echo color and then try to use the echo over an area filled with the other value, you will not be able to see it.

Effective Use of Color

It is beyond the scope of this manual to provide an exhaustive guide to color use; however, a few comments can be made on using color effectively. This section discusses seeing color first, to lay the groundwork. This is followed by a discussion on designing effective display images, since effective color use is almost impossible if the image is fundamentally unsound. Then, effective color use is discussed from both the objective and subjective standpoints.

Seeing Color

The human eye responds to wavelengths of electromagnetic radiation from about 400 nm to about 700 nm (4000 to 7000 angstrom.) We call this visible light. Visible light ranges from violet (400 nm) to red (700 nm.) If all the frequencies of visible light are approximately equally mixed, the result is called white light.

Two things seem to be associated with the sensitivity of the eye to various colors:

- The eye can distinguish the widest range of colors in the yellow-green region, and the smallest variety of colors in the blue region.
- The eye is most sensitive to detail in the yellow-green region.

It's All Subjective Anyway

No two people seem to perceive color the same way. In fact, the same person may perceive color differently at different times. In addition to the physiological and psychological variables in color perception, many environmental factors are important. Ambient lighting and surrounding color affect the perceived color tremendously.

Mixing Colors

If two distinct audio tones are played simultaneously, you will hear both of them. If the same area is illuminated by two or more different colors of light, you will not perceive the original colors of light, but rather a single color, and it will be not be one of the original colors. What you will perceive is called the **dominant wavelength**.

The CRT uses three different colored phosphors (red, green, and blue) and mixes various intensities of the resulting lights to produce one of X colors at any point on the CRT. What you actually see is the resulting dominant wavelength. This is an additive color system.

Mixing with pigments is a little different. Pigments in inks and paints absorb light. The idea with pigments is to subtract all but the color you want out of a white light source. This is a subtractive color system, and the primary colors are cyan, magenta, and yellow.

The different mechanisms for mixing additive and subtractive colors make it difficult to reproduce images created with additive colors (like a CRT) in a subtractive medium (like a plotted or printed page.) Photographing the CRT is the best method currently available for color hard copy. This problem is discussed in more depth later in this chapter under "Color Gamuts" and "Color Hard Copy."

Designing Displays

While the design of displays is not really a color topic, if the design of an image is fundamentally unsound, good color usage is not going to help it.

Whenever you put an image on a CRT, you have created a graphic design. The design will either be a good one or a bad one, and if you know this, you have automatically increased your chances of creating a good design. If you are going to create a lot of displays, you need a graphic designer. Many people have a natural talent for graphics —an ability to look at an image and tell whether it is graphically sound or not. If you don't have that talent (or feel you could use some help) there are two courses of action that might be productive for you: you can hire a graphics designer or become one. Hiring one is expensive and becoming one is time-consuming. However, if you are trying to communicate with users, *you must understand graphic design*. While getting a degree in graphic arts may be impractical, a course or two in the field will prove useful if you do much programming.

This manual can't turn you into a graphic designer, but a few simple hints may help you on your next program.

The most important thing in communicating with people is to keep it simple. Don't try to communicate the total sum of human knowledge in a single image. It is much more effective to have several screens of information that a user can call up as required than a single screen so complicated that the user can't find what he wants on it.

Try to redundantly encode everything in case one of the encoding methods fails. For example, if you color code information, use positional coding (the location of the information tells something about the nature of the information) too. Remember, the person reading the screen is probably *not* the person who wrote the program, and even if you are writing the program for yourself, you may forget how it works by the next time you try to use it.

Another important thing to remember is to be consistent. Always try to place the same type of information in the same area of the CRT and use the same encoding methods for similar messages. Don't using flashing to encode important information on one display and then using inverse video for the same thing seven displays into the program.

Objective Color Use

In spite of the subjectivity of color, there are some fairly objective things that you should know about color.

Color Blindness

The most common form of color blindness is red-green color blindness (the inability to distinguish red and green). Avoid encoding information using red-green discrimination, or these people will have difficulty using the system.

Color Map Animation

Motion is a powerful communication tool. Some simple forms of animation can be achieved by changing the colors in the color map. This technique is capable of adding simple motions to an image. Color map animation can be combined with frame buffer animation, which is based on creating images and storing them, to produce more dramatic animated effects.

The basic technique of color map animation can be broken into three steps:

1. Create the palettes (or starting palette.)
2. Create the image.
3. Load or modify the palette to add motion.

The program called RIPPLES on the *Manual Examples Disk* is an example of color map animation.

Subjective Color Use

Choosing appropriate colors for a program to use can be tricky, and constitutes a significant part of the job of a good graphic designer. If your application is complex, consult a graphic designer about the color scheme and layout of information displays for your program. Here are a few fundamental things to remember in designing your programs.

Choosing Colors

- First, and probably most important, use color sparingly. Color always has a communication value and using it when it carries no specific information adds noise to the communication.
- Use some method to select the colors—one of the best is a color wheel (see the SET PEN entry in the *HP BASIC Language Reference*).
    - Try varying the luminosity or saturation of a color and its complement (opposite it on the color wheel).
    - Try color triplets (three equally-spaced colors) and other small sets of colors equally-spaced around the color wheel.
- Pastels (less than fully-saturated colors) tend not to clash.
- Give careful attention to your background color. Remember that a filled area can become the background color for a portion of the image on the CRT.
    - If you are using a small number of colors, use the complement of one of them for the background.
    - If you are using a large number of colors, use a gray background.
- If two colors are not harmonious, a thin black border between them can help.
- Use subtle changes (such as varying the saturation or luminosity of a hue) to differentiate subtly different messages, and use major changes (such as large changes in the hue of saturated colors) to convey major differences.
- Most of all, think and experiment. The final criteria is "Does this display communicate the message?"

Color Spaces

If you ask a broadcast engineer what the primary colors are, he will probably tell you "Red, green, and blue." If you ask a printer what the primary colors are, he will probably tell you "Cyan, magenta, and yellow." If you ask a physicist, he will probably smile and say "That's not the right question." Let's begin with a closer look at color systems.

Primaries and Color Cubes

Actually, there are two sets of color primaries. Red, green and blue are additive primaries. Cyan, magenta, and yellow are subtractive primaries. Each of these sets of primaries can be used to construct what is referred to as a **color cube**. These are called the RGB color cube and the CMY color cube.

Each of the color cubes can be used to describe a **color space**. Color spaces are mathematical abstractions which are convenient for scientific descriptions of color. This is because the color spaces provide a coordinate system for describing colors. Once you have a coordinate system, you can talk about and manipulate colors mathematically.

In addition to the color cubes, other color coordinate systems exist. While there are many, we will only look at HSL color space, because it is one of the available color models on your color computer. First, the cubes.

The RGB Color Cube

The RGB color cube describes an **additive** color system. In an additive color system, color is generated by mixing various colored light sources. (Color mixing is discussed in "Effective Use of Color.")

The origin (0,0,0) of the RGB color cube is black. Increasing values of each of the additive primaries (red, green, and blue) move towards white (the opposite corner of the cube.) The maximum for all three colors is white (1,1,1).

A diagonal of the cube connecting (0,0,0) and (1,1,1) represents gray shades, which are generated by incrementing all three color axes equally.

The RGB color cube can be accessed directly, in 16 steps (4-plane systems) or 256 steps (8-plane systems) for each axis, by the INTENSITY option for the color definition statements (SET PEN, AREA INTENSITY, and AREA COLOR).

The CMY Color Cube

The CMY color cube represents a **subtractive** color system. In a subtractive color system, colors are created by subtracting colors out of a pure white (containing all colors equally) light source. This most often occurs when light is reflected off surfaces containing or coated with pigments. This happens in printing and painting, among other operations.

The origin (0,0,0) for the CMY color cube is white. This represents all the colors in a perfect white (containing all colors) light source being reflected by a white (reflecting all colors) surface. Increasing values of each of the subtractive primaries (Cyan, Magenta, and Yellow) move towards black (the opposite corner of the cube). The maximum for all three colors is black (1,1,1).

A diagonal of the cube connecting (0,0,0) and (1,1,1) represents gray shades, which are generated by incrementing all three color axes equally.

Converting Between Color Cubes

It is sometimes useful to convert from one color coordinate system to another. The CMY color cube can be converted to RGB coordinates (or RGB to CMY) by producing a color triplet (a 1 $\times$ 3 matrix) containing the CMY coordinate and subtracting this from a color triplet representing a unit color vector (1,1,1). This operation represents rotating the color cube to bring the CMY black (1,1,1) to the RGB black (0,0,0).

The following program lines convert the RGB color map into CMY values. This is done to provide separations of an RGB image into CMY values for printing (remember–printing is a subtractive process). Since the system is color mapped, you only need to convert 16 (4-plane systems), 64 (6-plane systems) or 256 (8-plane systems) values—remember, the frame buffer values only point to a register in the color map.

- The contents of the color map are copied into `Old_colors` using a GESCAPE in line 14680.
- Each color triplet in the color map is copied into `Rgb_point` in lines 14720 through 14740.

- The actual RGB to CMY conversion is done in line 14750.
- The CMY triplet is copied into the CMY array in lines 14760 through 14780.

```
14660 Convert_colors:!
14670   ALPHA ON
14680   GESCAPE 3,2;Old_colors(*)
14690   PRINT "       OLD COLORS               NEW COLORS"
14700   PRINT "Index  R     G     B            C     M     Y"
14710   FOR I=0 TO 15
14720     FOR J=1 TO 3
14730       Rgb_point(J)=Old_colors(I,J)
14740     NEXT J
14750     MAT Cmy_point= Unit_point-Rgb_point
14760     FOR J=1 TO 3
14770       New_colors(I,J)=Cmy_point(J)
14780     NEXT J
14790     PRINT USING Image$;I,Rgb_point(1),Rgb_point(2),Rgb_point(3),
          Cmy_point(1),Cmy_point(2),Cmy_point(3)
14800   NEXT I
14810   Converted=True
14820   RETURN
```

A subprogram can be used to provide drivers to produce monochromatic gray-scale displays representing the cyan, magenta, and yellow contents of the color map (and a separate black image that printers like to have around). The monochromatic representation is easier to photograph than the actual color content.

The color conversion just described is mathematical. If you really want to print it, you will have to work with a printer to calibrate the frames you are giving him against a good color photo of the actual image. The printer may also want the CMY information to be inverted for his process. This can be achieved photographically or by subtracting each of the CMY values from one during the color map conversion (this is an element-by-element subtraction, not a matrix subtraction). The conversion can be achieved easily with the MAT statement:

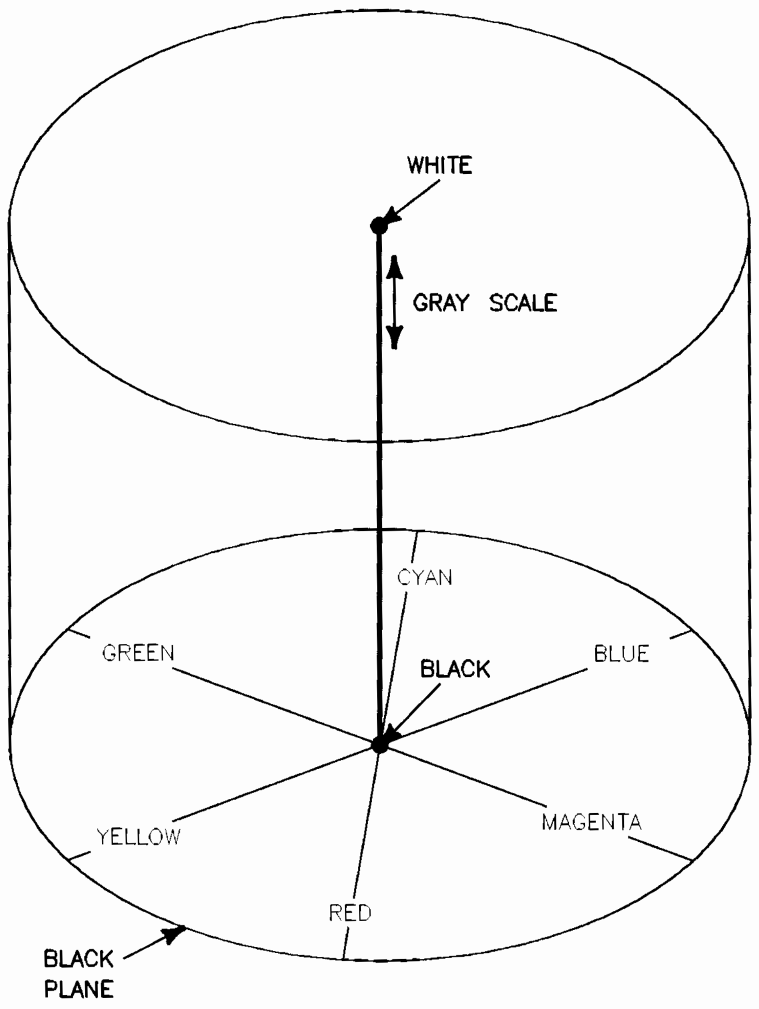
```
14805   MAT New_colors = (1) - New_colors
```

HSL Color Space

Color cubes are useful for working with physical systems based on color primaries. They are not always intuitive, though.

The HSL color cylinder resides in a cylindrical coordinate system. A cylindrical coordinate system is one in which a polar coordinate system representing the X-Y plane is combined with a Z-axis from a rectangular coordinate system.

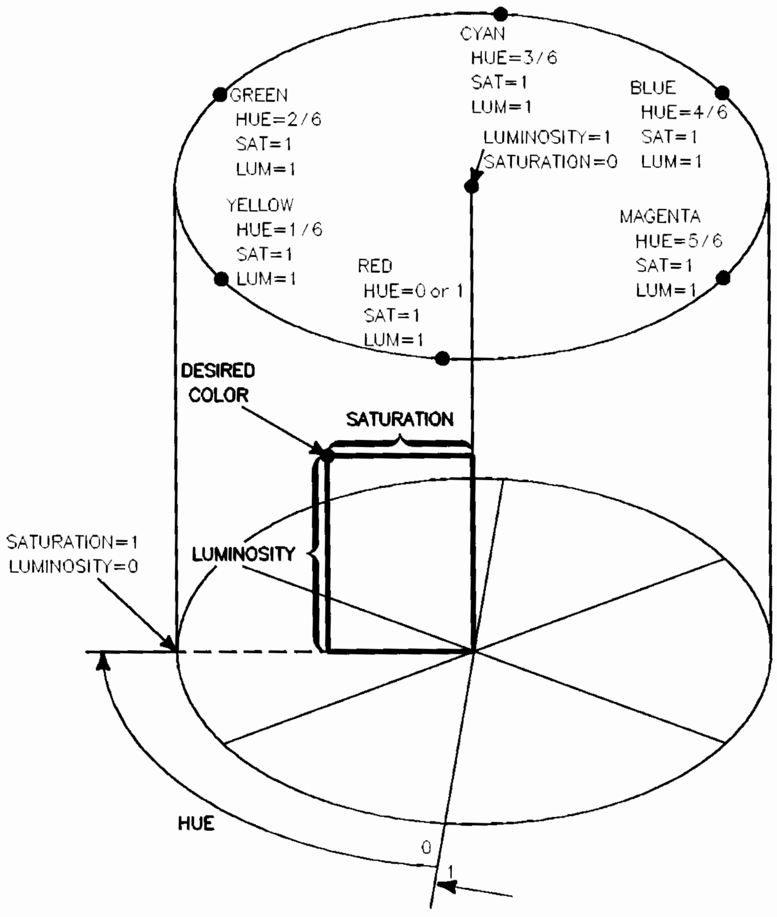
- The coordinates are normalized (range from 0 through 1).
- *Hue* (H) is the angular coordinate.
- *Saturation* (S) is the radial coordinate.
- *Luminosity* (L) is the altitude above the polar coordinate plane.



HSL Color Cylinder

The cylinder rests on a black plane (L = 0) and extends upward, with increasing altitude (Luminosity) representing increasing brightness. Whenever luminosity is at 0, the values of saturation and hue do not matter.

White is the center of the top of the cylinder (L=1, S=0). The center line of the cylinder (S = 0) is a line which connects the center of the black plane (L=0, S=0) with white (L=1, S=0) through a series of gray steps (L from 0 to 1, S=0). Whenever saturation is 0, the value of hue does not matter. The outer edge of the cylinder (S=1) represents fully saturated color.



HSL Color Specification

Using the above drawing (HSL Color Specification,) hue is the angular coordinate, saturation is the radius, and luminosity is the altitude of the desired color.

HSL to RGB Conversion

Converting from HSL to RGB is simple. Do a SET PEN for the HSL point you want and then read it out of the color map with a GESCAPE. You are limited to the resolution of the color map, but it is very simple. The following line reads the color map into `Old_colors`.

```
14680   GESCAPE 3,2;Old_colors(*)
```

RGB to HSL conversion is not described, due to the fact that it is a one-to-many conversion (the entire bottom plane of the HSL color space is represented by a single point in the RGB color space, and hue is indeterminate if saturation equals 0).

Color Gamuts

The range of colors a physical system can represent is called its **color gamut**. Color gamuts are important when you want to convert between different physical systems because the source system may be able to produce colors the destination system cannot reproduce. An exhaustive treatment of color gamuts is beyond the scope of this book. However, here are some rules of thumb:

- The color gamuts for CRTs and photographic film are not the same, but are fairly close. If you are lucky, you can photograph the CRT and catch it on film. It may take more than one exposure, so be careful and bracket everything with several exposures.
- The color gamut for printing is significantly smaller than that of either photographic film or of a CRT. The fact that you have a picture of a CRT does not mean you can hand it to a printer and get a faithful reproduction of it.
- The color gamut of a plotter is much smaller than that of a CRT. You have to create images with the limitations of a plotter in mind if you intend to reproduce them on a plotter (see "Plotting and the CRT," below.)

Color gamuts are not a problem unless you forget the differences and try to act like all physical systems have the same gamut. Think ahead if you have to reproduce images–it will save a lot a trouble.

Color Hard Copy

Color hard copy represents a translation between color systems, and many of the problems in color hard copy arise from the fact that the color gamuts available to the CRT and the hard copy device are different.

There are three basic ways to get a color hard copy of what is displayed on a color computer:

- Take a photograph of the CRT.
- Re-run the program that generated the image with an external plotter selected as the display device (`PLOTTER IS 705,"HPGL"`).
- Use the PaintJet^{TM} printer.

The first method is the easiest and can capture (usually) whatever is on the CRT, regardless of what colors are used (see "Color Gamuts," above.) The second requires setting up the color map to match the pens in a plotter, and is not as likely to capture what you see on the screen. The third method requires the PaintJet^{TM} color printer and the GDUMP_C Utility (on the Utilities 2 Disk). See the "BASIC Utilities Library" section of the *Installing and Maintaining HP BASIC* manual for information about GDUMP_C.

Photographing the CRT

Photography is an art, not a science. Capturing images off a CRT is relatively straightforward, but sometimes unpredictable due to the different color gamuts available for film and the CRT. The following guidelines will provide a starting point. If your images are not "typical", you may have to go back and re-photograph some of them.

- Use ISO 64 Color film.
- Set up your equipment in a room that can be darkened.
- Use a telephoto lens (the longer the better, up to about 500 mm) to minimize the effects of the curvature of the CRT.
- Use a tripod.
- Darken the room and take a one-second exposure.
- Bracket the aperture around *f*5.6. (One stop above and below.)

Plotting and the CRT

There are two basic reasons the CRT is hard to capture on a plotter.

- The CRT is an additive color device and a plotter is a subtractive color device.
- The color gamut of the CRT is much larger than that of the plotter.

The conversion from additive to subtractive colors is not a huge problem if the plot is a simple line drawing with few intersections and area fills. If the plot is complex, especially with lots of intersections and overlapping filled areas, the plot is much less likely to capture the display image accurately.

A possible technique described below *purposely* limits the color gamut of the CRT to give the plotter some chance of capturing it.

To set up the color map and plotter to match one another:

- Set your background to white (`SET PEN 0 INTENSITY 1,1,1`).
- Set up pens matching the color map colors in slots 1 through 8 in the same order they are presented in the default color map listed under "Default Colors."
- Use pen selectors from 8 through 15 to select your pens.
- Run the program with the color mapped CRT as the display device, modifying it as necessary to produce the image you want on the CRT.
- Re-run the program with the plotter as the display device. You will need to subtract 8 from the pens to properly select the set available on the plotter.

While it is possible to get some idea of the plot that will be produced on the plotter, don't be surprised if they don't look exactly the same.

Color References

The following references discuss color and vision. Texts that serve as useful introductions to the topic are starred.

* Cornsweet, T., *Visual Perception.* New York: Academic Press, 1970

Farrell, R. J. and Booth, J. M., *Design Handbook for Imagery Interpretation Equipment.* (AD/A-025453) Seattle: Boeing Aerospace Co., 1975

Graham, C. H., (Ed.) *Vision and Visual Perception.* New York: J. Wiley & sons, Inc., 1965

* Hurvich, L. M., *Color Vision: An introduction.* Sunderland, MA: Sinauer Assoc., 1980

Judd, D. B., *Contributions to Color Science.* (Edited by D. MacAdam; 545) NBS special publication Washington: U. S. Government Printing Office, 1979

* Rose, A., *Vision: human and electronic.* New York: Plenum, 1973

Using Gray-Scale Displays

The "internal" display controller of the HP 9000 Model 362, 382, and Series 700 computers supports both color and gray-scale displays. Let's consider what a gray-scale display is, and how it differs from a color display or a simple monochrome display:

- A *monochrome* display has pixels of only one color (perhaps amber, green, or white) and displays images by turning pixels either fully "on" or "off." There are no "shades of gray" (or amber or green), which limits your ability to display graphic images.
- A *color* display has pixels of the three primary colors (red, green, and blue). By varying the luminosities of the pixels of each color, the display can produce a continuous-tone color image.
- A *gray-scale* display, like a monochrome display, has pixels of only one color. However, like a color display, the gray-scale display can vary the luminosity of each pixel to produce a continuous-tone image. If the display has "white" pixels, the continuous-tone image can be described as "shades of gray,"

and thus, the display is known as a "gray-scale" display. Although some "gray-scale" displays have amber, green, or some other color of pixels, they all produce continuous-tone images and we will use the term "gray-scale" to describe them.

Gray-Scale Modes of Operation

Since gray-scale displays produce a range of luminosities, from the programmer's standpoint they behave very much like color displays. In fact, gray-scale displays, like color displays, function in both "color-mapped" and "non-color-mapped" modes. (However, if you are running BASIC/UX in the X Windows environment, the non-color-mapped mode is not supported.) BASIC automatically determines whether you have a gray-scale or color display when booted.

Note If you boot BASIC with a color display connected, the default PEN value is 1 (white) and the default ALPHA PEN value is 4 (green). However, if a gray-scale display is connected, both the PEN and ALPHA PEN values are 1 (luminosity = 1) by default. This ensures that characters are readable.

Non-Color-Mapped Mode

To specify the non-color-mapped mode, execute:

```
PLOTTER IS CRT,"INTERNAL"
```

just as for a color display.

The following table lists the luminosity (brightness) values for pens 0 through 7 in non-color-mapped mode.

Table 5-1. Non-Color-Mapped Gray Scale

Pen Number	Luminosity
0	0
1	1
2	0.30
3	0.89
4	0.59
5	0.70
6	0.11
7	0.41

The luminosities for pens 0 through 7 are derived from the luminosities of the corresponding colors that a color display would use in non-color-mapped mode.

In non-color-mapped mode, pens 8 through 255 merely repeat the luminosity values of pens 1 through 7 over and over again.

Note The HP Measurement Coprocessor also supports the use of gray-scale displays. In the measurement coprocessor "NTSC" mode, pens 0 through 7 follow the same luminosity values given in the above table. However, pens 8 through 255 are not supported for the measurement coprocessor. For further information, refer to *Installing and Using HP BASIC/DOS*.

Color-Mapped Mode

To specify the color-mapped mode, execute:

```
PLOTTER IS CRT,"INTERNAL";COLOR MAP
```

just as for a color display.

In color-mapped mode, the gray-scale luminosities for pens 0 through 15 correspond to the standard CIE (NTSC) gray-scale values developed for black and white television. The following table lists the luminosity (brightness) values for pens 0 through 15.

Table 5-2. Color-Mapped Gray Scale

Pen Number	Luminosity
0	0
1	1
2	0.30
3	0.89
4	0.59
5	0.70
6	0.11
7	0.41
8	0
9	0.69
10	0.51
11	0.47
12	0.44
13	0.56
14	0.58
15	0.60

The luminosities for pens 0 through 15 are derived from the luminosities of the corresponding colors that a color display would use in color-mapped mode.

In color-mapped mode, pens 16 through 255 provide a continuously descending range of gray luminosities. Use the following formula to determine the luminosity value for any PEN value in the range 16–255.

$$Luminosity = \frac{256 - PEN}{241}$$

Graphics Programming for Gray-Scale Displays

As mentioned previously, when BASIC is booted it will automatically determine whether a color or gray-scale display is connected. Thus, when you specify "non-color-mapped" or "color-mapped" mode, the appropriate color scale will be selected for a color display, or the appropriate gray scale will be selected for a gray-scale display. This ensures that a graphics program written for a color display will also work for a gray-scale display. A program that produces a good image on a color display should also produce an acceptable image on a gray-scale display. However, there are some "tricks" that will help you improve the image on the gray-scale display.

Let's start by looking at the formula that BASIC uses to translate RGB color values to gray luminosities:

$$Luminosity = 0.3 * R + 0.59 * G + 0.11 * B$$

Obviously, any green in the color image will be weighted the most in the gray-scale image, while any blue will be weighted the least. A practical example will clarify this.

Suppose that you have written a program that displays an ocean view. The sky is shown as light blue on the color display, and the ocean as a slightly darker blue. On the color display you probably won't have any trouble distinguishing the sky from the ocean. However, on a gray-scale display, you may not be able to tell the difference. The reason? Since blue is only weighted 11 percent in the gray-scale luminosity calculation, the difference in luminosity is too small. To solve this problem, you will need to use a darker blue for the ocean, and a lighter blue for the sky. Thus, you can produce an image that works on both displays.

The best thing to do is experiment with pen values and luminosities to find the best solution in each case. Of course, if you don't need to display the image on a color display, you can simply optimize it for the gray-scale display.

6

Graphics Input and Display

This chapter discusses interactive graphics, graphics input devices, and some advanced methods of displaying data.

Interactive Graphics

The high speed of graphics operations on Series 200/300/400/700 computers makes possible a powerful mechanism for communicating with the computer: *interactive graphics*. One way to understand interactive graphics is to see it in action. If your computer has a knob or mouse, LOAD and RUN the program `BAR_KNOB`, from your *Manual Examples* disk.

Note

The *Manual Examples* disk (`/usr/lib/rmb/demo/manex` for BASIC/UX users) contains programs found in this chapter. As you read through the following sections, load the appropriate program and run it. Experiment with the programs until you are familiar with the demonstrated concepts and techniques.

Both HP BASIC/UX 300/400 and HP BASIC/UX 700 include a number of BASIC programming examples in `/usr/lib/rmb/demo/manex` which may not perform as documented in this chapter. Some of those examples illustrate programming techniques for obsolete computers and peripherals. However, you can easily leverage the techniques presented in those examples to develop programs that do work on your version of HP BASIC/UX. Just be aware that, in some cases, hardware dependencies might produce strange or unexpected results.

If you turn the knob clockwise, or move the mouse in one direction, the bar graph displayed on the screen will indicate a larger value. At the same time, the numeric readout underneath the bar will increase its value. Turning the knob counterclockwise, or moving the mouse in the opposite direction, has the opposite effect. This program demonstrates the key characteristics of an interactive graphics system.

Elements of an Interactive Graphics System

The minimum elements of this type of graphics system are as follows:

- A graphic display that represents the contents of the data structure.
- An input mechanism for interacting with the displayed image (the knob or mouse, in this case.)
- A data structure. (The value displayed underneath the bar is the contents of a variable that we are modifying. The BASIC variable containing the value is a simple data structure.)

Interactive graphics can be as simple as representing a single value on the screen and providing the user a method for interacting with it. It can also be as complex as a printed circuit layout system. This chapter does not tell you how to build a printed circuit layout system, but it does provide some hints on implementing interactive graphics systems that work.

Characterizing Graphic Interactivity and Selecting Input Devices

One important aspect of designing a good interactive graphics system is correctly characterizing the interaction with the system. Properly characterizing the interactivity allows you to select the most appropriate device for interacting with the system. To characterize the interaction, consider the following:

- The number of *degrees of freedom* in the system. This is the number of ways in which a system can be changed.

- The *quality* of each of the degrees of freedom. This describes how the input to a degree of freedom can be changed.
- The *separability* of the degrees of freedom.

The BAR_KNOB program has limitations in these regards. Read on to see why.

Single Degree of Freedom

Many interactive graphics programs need deal only with a single degree of freedom. The appropriate control device for such programs depends on whether continuous control or quantizable control is needed.

The program BAR_KNOB is a good example of a continuous, single degree of freedom. The knob is ideal for controlling a program like this. If "fine tuning" is needed, the shift key can be used as a multiplier to change the interpretation of the knob. It is also possible to use the softkeys for fine tuning.

Softkeys can be used for quantizable control of a degree of freedom. It is also possible to use keyboard entry of numeric values for quantizable information.

Non-Separable Degrees of Freedom

One characteristic of multiple, non-separable degrees of freedom is that they are generally continuous. The most common operation of this type is free-hand drawing. This is most easily accomplished with a graphics tablet.

Separable Degrees Of Freedom

In many programs, the degrees of freedom are completely separable. In fact, for some operations, it is preferable to have totally independent control of the degrees of freedom of the model.

All Continuous

If all the degrees are continuous, a good choice is using the softkeys to select the degree of freedom and then using the knob to control the input to that degree of freedom. An even better choice is to use an HP 46085 Control Dial Box, which has nine knobs. (Run the program in file `CDials` to see an example.)

All Quantizable

If all the degrees are quantizable, using softkeys is ideal.

Mixed Modes

In most sophisticated graphics systems, several degrees of freedom in the system interact with each other. A good example is a graphics editor. In a graphics editor, your primary interaction is with a visual image, and the degrees of freedom (X and Y location) for that operation are partially separable, at best. (They are non-separable if the system supports freehand drawing.) There is also a degree of freedom involved in controlling the program. The program control is strongly separable from image creation.

The most appropriate device for supporting mixed modes is a graphics tablet. The HP 9111A tablet supports two modes of interaction by partitioning the digitizing surface into two areas. Sixteen small squares along the top of the tablet are used as softkeys to provide a control menu. The large, framed area underneath the softkeys is the active digitizing area. The active digitizing area is used for interacting with the image you are creating. Some HP-HIL tablets (such as the HP 46087A and HP 46088A) use a 4-button stylus, or "puck," which has physical buttons on the cursor device.

It is possible to combine the quantized, separable control operations with continuous, non-separable image editing. This is done by using the active digitizing area for interacting with the image and using the menu area for controlling the operations available in the editing program. The operator does not have to change control devices to access the different interaction modes.

Echoes

An important part of interactive graphics is letting the operator know "where he is at." This can be done by updating the image (as in BAR_KNOB). In other operations—such as menu selection, object positioning, and freehand drawing—it is important to show the operator where he or she is. In many cases, this can be done with the SET ECHO statement.

The Built-in Echo

Many graphics applications can be handled using the built-in echo. The following program continuously tracks the input locator and monitors the pressed/not-pressed status of the Digitize button (or stylus). The cursor position is continuously echoed on the output device, and lines are drawn if the Digitize button (or stylus point) is pressed.

```
100      GINIT                                  ! Restore defaults
110      GRAPHICS INPUT IS 706,"HPGL"           ! Define input
120      PLOTTER IS CRT,"INTERNAL"              ! Define output
130      GRAPHICS ON
140      VIEWPORT 0,133,0,100                   ! Match aspect ratios
150      WINDOW 0,100,1,100                ! Define UDUs
160      FRAME                                  ! Draw limits
170      !
180      LOOP
190        READ LOCATOR X,Y,Status$
200        SET ECHO X,Y
210        Button$=Status$[1,1]
220        GOSUB Action
230      END LOOP
240      !
250 Action:    IF Button$="0" THEN MOVE X,Y
260             IF Button$="1" THEN DRAW X,Y
270             RETURN
280            !
290      END
```

This program, as it stands, will only work with an HP 9111 graphics tablet at address 706. If you wish to use a mouse or an HP-HIL tablet, change the GRAPHICS INPUT IS statement to:

```
GRAPHICS INPUT IS KBD,"KBD"
```
(for HP-HIL Mouse)

or

```
GRAPHICS INPUT IS KBD,"TABLET"
```
(for HP-HIL Tablet)

Making Your Own Echoes

In some applications, the cross-hair generated by SET ECHO is not sufficient. You may want to generate a *rubber band* line or box. A rubber band line stretches from an anchor point to the echo position. In these cases, it is necessary to draw your own echo.

Since an echo needs to be repositioned as the operator interacts with it, it must be constantly drawn and redrawn. If it is just drawn and then erased, the background it is drawn over will soon become littered with erased images of the echo. What we really want to do is find a way to draw it and then "undraw" it, rather than erasing it. The complementary drawing mode is used to do this. In the complementary drawing mode, the bits specified by the current pen selector are complemented in the frame buffer, rather than just overwriting the contents. If a second complement is done, the image is restored to whatever was there before the echo was written to it. The echo generated by SET ECHO is automatically drawn in the complementary mode.

It is important to remove any echo you have drawn on the screen *before* updating the image. Complementing a bit pattern does not restore the image if the image was altered between the complementary drawing and undrawing. This is done automatically by SET ECHO, but you must handle it yourself if you are building your own echoes. The following loop will support a tablet with several different echoes when used with the echo routines discussed below.

```
570   LOOP                                 ! Main Tracking Loop
580     READ LOCATOR Xin,Yin
590     DISABLE
600     CALL Make_echo(Xin,Yin,Echo_type)  ! Several Echo Types
610     ENABLE
620   END LOOP
```

Two sets of echo routines (Kill_echo and Set_echo) are provided, one for monochrome and one for color systems.

Monochrome Echoes

Select Pen 0 to access the complementary drawing mode. The subroutines in the example program Graph_echo on the *Manual Examples Disk* demonstrate the techniques of rubber band line and rubber band box echoes. Be aware that these subroutines would be part of some larger program that you create. Run Graph_echo to see this example.

Color Echoes

In color, accessing the complementary drawing mode is slightly different. To access the complementary drawing mode, specify a negative pen number after a GESCAPE to select the non-dominant writing mode (operation selector of 5). The subroutines implement rubber band lines, and have hooks in place for rubber band boxes. Study the subroutine listing in file Graph_echo to examine the programming techniques used.

Graphics Input

In many interactive graphics applications the tablet is used as an echo mover. The transformation between the graphics tablet and the display should be linear in such applications, but the axes do not have to transform through the same scaling. It doesn't matter if a square on the tablet represents a square on the display if you are just using the tablet to move a cross-hair on the display. However, if you are trying to copy an image from paper to the display (using a graphics tablet) it is important to preserve both the linearity and the aspect ratio in the transformations.

The **maximum usable area** of a graphics device is bounded by its hard clip limits. For example, the pen cannot be made to draw outside these limits on an output device.

The **current usable area** is bounded by the rectangle defined by the points P1 and P2; the lower-left corner is P1, and the upper-right corner is P2. On many devices, these points can be moved manually or by the program.

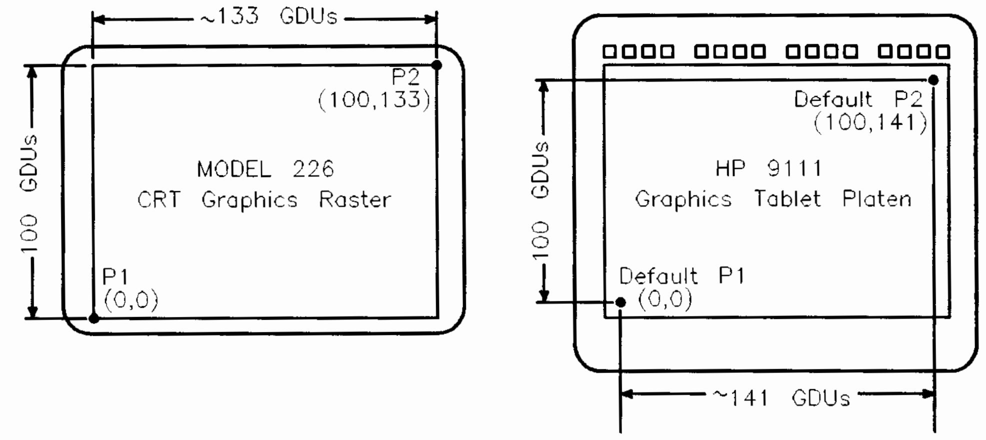
When a GINIT or PLOTTER IS statement is executed, points P1 and P2 are read from the plotting device. With GINIT, the plotting device is assumed to be the internal CRT. The value of RATIO is then set to the result of the following calculation:

$$RATIO = \frac{(P2x - P1x)}{(P2y - P1y)}$$

GINIT does a WINDOW for the internal CRT only; PLOTTER IS does not do implicit windowing. You must explicitly do the following statements:

If RATIO $\geq$ 1:	`VIEWPORT 0,100*RATIO,0,100`
	`WINDOW 0,100*RATIO,0,100`
If RATIO $<$ 1:	`VIEWPORT 0,100,0,100/RATIO`
	`WINDOW 0,100,0,100/RATIO`

As seen above, both the X and Y coordinates of P1 are always 0 GDU. The default graphic display unit coordinates of P2 depend on the device; however, the smaller coordinate of this point is always 100 GDUs. Two examples are shown below:



Usable-Area Boundaries:

Left edge = X coordinate of P1 Right edge = X coordinate of P2

Bottom edge = Y coordinate of P1 Top edge = Y coordinate of P2

When a PLOTTER IS statement is executed, the locations of points P1 and P2 on the specified device are determined. The current VIEWPORT statement parameters define the physical area in GDUs, which is scaled in UDUs by the WINDOW or SHOW statement currently in effect.

When a subsequent GRAPHICS INPUT IS statement is executed, BASIC attempts to apply the current VIEWPORT and WINDOW (or SHOW) parameters to the P1,P2 rectangle of the input device. In the preceding example, the two usable areas are not identical in size (in GDUs), since the HP 9111 has a smaller horizontal-to-vertical aspect ratio. This difference in aspect ratios may produce *two types of potentially undesirable results* when using these two devices together for interactive graphics capabilities. The GRAPHICS INPUT IS sets the hard clip limits of the input device to the largest space possible that has the same aspect ratio as the output device.

HP-HIL Devices

The HP-HIL (Hewlett-Packard Human Interface Loop) family of peripherals is supported on the Series 200/300/400/700 computers. The HP-HIL graphics input devices include the knob, mouse, and graphics tablets. The `KBD` binary is required for all HP-HIL devices except keyboards when used on the Series 200/300 computers.

HP-HIL Relative Locators (Such as Cursor Keys, Knob, or Mouse)

The term **relative locator**, in the context of a graphics input device, refers to a device which returns incremental X,Y offsets. The computer uses these incremental values to update the logical coordinates of the graphics locator. No fixed physical reference exists.

Relative locators can also provide "keystrokes" to represent the various buttons, but this capability is not available while the relative locator is being used for graphics input.

The button that is pressed is reported to the main program by the seventh and eighth bytes of the status string returned by `DIGITIZE` and `READ LOCATOR`. To determine if any buttons were pressed, check `VAL(Status$)`. If the value is non-zero, a button was pressed. To determine *which* button was pressed, check the appropriate bit in the number represented by the ASCII characters in positions 7 and 8 of the string. For example:

```
ALLOCATE Button(0:Max_buttons)
   :
FOR Bit=0 TO Max_buttons
  Button(Bit)=BIT(VAL(Status$[7]),Bit)
NEXT Bit
   :
IF Button(<< n >>) THEN . . .
```

The keyboard is a relative device, also. For example, pressing (Enter) will trigger a `DIGITIZE` with a status string of "`1,2,0,00`". However, this does not show up in a `READ LOCATOR` operation.

Executing `GRAPHICS INPUT IS KBD,"KBD"` turns on *all* relative locators and the results are combined. Thus, you can safely intermix input from arrow keys, an internal knob, external knobs, and mice.

HP-HIL Absolute Locators (Such as a Tablet or TouchScreen)

An **absolute locator** has a finite mapping area, such as the HP 9111 tablet or the HP 35723A Touchscreen. On these devices, the data returned are X,Y coordinate pairs. Each possible value of these pairs corresponds to a fixed location on the physical surface of the digitizing device.

Proximity, in the following paragraphs, is defined as the area in which a locator can detect that you are pointing to something. For example, on a HP 35723A Touchscreen, proximity is where your finger is close enough to the screen that the computer can assign a location to your finger's "shadow." For a graphics tablet, proximity is where the tablet can detect the presence of the stylus or puck.

The Touchscreen is supported as a `GRAPHICS INPUT` device—a low-resolution `TABLET`. Going into proximity on a Touchscreen—pointing to something with your finger touching the screen—causes the Touchscreen to sense your finger's location. Going out of proximity on a Touchscreen—removing your finger from the screen—triggers a `DIGITIZE`. On the Touchscreen and the HP 45911A, HP 46087A and HP 46088A tablets, when out of proximity, bytes 7 and 8 of the "device status" string contain the ASCII characters "64".

The HP-HIL tablets consider all buttons not pressed when out of proximity. Buttons which are held down while moving into proximity are sent as key presses at the proximity transition. This can trigger a digitize at that point.

Similar to combining relative locators, executing GRAPHICS INPUT IS KBD,"TABLET" turns on all absolute locators and combines the results. However, since the display can be scaled to only one absolute locator at a time, and since the inputs replace each other (as opposed to adding to each other), this is not a useful feature. See the program in file KBD_ICONS for an example of a program that supports the use of all of these devices.

GESCAPE, in conjunction with the device selector KBD, allows you to set and read hard clip limits for absolute locators on the HP-HIL bus. Note that the hard clip limits are only the right-most and uppermost limits; the left and bottom edges of the plotting surface are always zero. For example, to set the hard clip limits for an HP-HIL Touchscreen in spite of the presence of a tablet on the bus:

```
10  INTEGER Parameter_array(1:2)
20  Parameter_array(1)=52
30  Parameter_array(2)=46
40  GESCAPE KBD,20,Parameter_array(*) ! Set absolute-locator hard limits
```

To read the hard clip limits for all the absolute locators on the HP-HIL bus, you can use GESCAPE with operation selector 21 or 22. For example:

```
10   INTEGER Param_array(1:4)  ! We have TWO absolute locators on the HP-HIL
20   GESCAPE KBD,22;Param_array(*)  ! Read ALL absolute locator limits
```

If Param_array had been larger than four elements, the first unused element—after using two elements for each absolute locator on the loop—would contain −1. This special value indicates that there are no more coordinate pairs.

Unlike other GESCAPE operation selectors, operation selectors 20 through 22 do not require the device at the specified select code to be currently active. Indeed, if you want to set the hard clip limits, GESCAPE KBD,20 must be executed *before* GRAPHICS INPUT IS KBD,"TABLET".

Operation selectors 20 and 21 will give `DEVICE NOT PRESENT` errors if no tablet, Touchscreen, or HP-HIL interface exists. An operation selector 22, under the same circumstances, will return a −1 for the first entry in the return array.

The HP-HIL tablets can be treated as a superset of HP 9111A when used with the built-in GRAPHICS INPUT IS, READ LOCATOR, and DIGITIZE commands. The HP-HIL tablets are a superset because of the extra button information available in the READ LOCATOR and DIGITIZE status strings.

There is one important difference between the HP 9111A and the HP-HIL tablets which may cause programs that work with the HP 9111A to have problems with the HP-HIL tablets. When a READ LOCATOR or DIGITIZE command is executed for an HP 9111A, BASIC sends a request to the HP 9111A for the current location of the stylus. The HP-HIL tablets, on the other hand, send the current location only when it changes. They cannot be asked for the current location. When `GRAPHICS INPUT IS KBD,"TABLET"` is executed, the internal variables representing the state of the HP-HIL Tablet are initialized. Since BASIC does not know the true state of the tablet at the time, the initialization sets up a unique state which can become valid with any set of received data for the tablet, but which can also be easily recognized as invalid. This state is in proximity with negative device coordinates.

This difference should not be a problem for a program which executes the GRAPHICS INPUT IS statement initially and leaves graphics input active while it runs. However, it may be a problem if the program was written to do a GRAPHICS INPUT IS before each call to READ LOCATOR because this causes initialization to occur just before input. Note that after `GRAPHICS INPUT IS KBD,"TABLET"` is executed, the data returned by READ LOCATOR will be recognizably invalid until a transaction on the tablet causes valid data to be sent to the controller.

Support of HP-HIL Devices for Graphics Input

Here is a generic description of the HP-HIL devices for which BASIC provides drivers:

- Relative locators with one to three dimensions and up to six buttons;
- Absolute locators with one or two dimensions and up to six buttons and/or proximity.

You can write your own drivers for other devices. See the "HP-HIL Interface" chapter of the *HP BASIC Interface Reference* manual.

For both of these categories of devices, extra dimensions or multiple sets of axes disqualify the device. Note that the maximum number of buttons is six; any "seventh" button would be ignored.

Note BASIC only configures the HP-HIL bus at power-up and `SCRATCH A`. Reconfiguring the bus physically without doing a `SCRATCH A` can result in devices not being recognized and/or in data being misinterpreted as coming from another type of device. On HP-UX, a new X Windows session must be initiated by exiting the current session and starting a new one.

Dealing With Multiple Buttons

The HP 46060A mouse and the HP 46089A digitizer puck have multiple buttons available to press when digitizing. This means that you can choose, when digitizing, how to signal to the computer that you've made your choice.

The computer finds out which button you pressed by the status string returned from `DIGITIZE` and `READ LOCATOR`. The example program in the file `KBD_ICONS` (on the *Manual Examples Disk*) tracks, on the CRT, the stylus movements on the digitizer. Note the use of this status string in this program.

Menu-Picking

Perhaps one of the most common uses for the Touchscreen is that of presenting several options on the screen and having the user select one by pointing to it. The example program `KBD_ICONS` presents five options on the screen, and the user picks one. The main program then states which option the user picked. You may enhance and modify the example to fit your application.

Points to note in the example:

- The `FNMenu` is a general-purpose function to which a menu array can be passed, and from which the selected option is returned.
- The user must indicate a location *within* the option boxes; if not, a warning will be given, and the user must try again. This prevents invalid data from being returned to the calling routine.
- The options are `LABEL`ed in graphics, rather than being `PRINT`ed in alpha. This avoids the difficulty in aligning alpha text and graphics scaling, which is where `DIGITIZE` works from.

Selecting a Graphics Input Device (BASIC/UX only)

This section discusses the use of the statement GRAPHICS INPUT IS for assigning the graphics input device in a window environment with BASIC/UX.

The GRAPHICS INPUT IS Statement

6

The statement GRAPHICS INPUT IS defines which device is to be used for graphics input in subsequent DIGITIZE, SET LOCATOR, READ LOCATOR, and TRACK IS ... ON/OFF commands. The syntax for this statement is as follows:

`GRAPHICS INPUT IS` *device_selector*`,"`*digitizer_specifier*`"`

where *device_selector* may be KBD, or the select code and primary address of a graphics input device (e.g., 706). The *digitizer_specifier* must be one of the following:

`KBD` or `ARROW KEYS`	Specifies relative pointing devices such as the cursor keys, knob, or mouse.
`TABLET`	Specifies absolute pointing devices such as HP-HIL tablets and the Touchscreen.
`HPGL`	Specified if the device selector is anything other than the keyboard select code.

Special Considerations

This section covers considerations when selecting:

- a relative graphics input device in X Windows (such as an HP-HIL Mouse)
- an absolute graphics input device in X Windows (such as an HP-HIL A-size Digitizer)
- an HPGL device with multiple BASIC/UX processes (such as an HP 9111A Digitizer).

It also covers considerations for terminal keyboards (such as an HP 2393A) as graphics input devices.

If the information in this section is not considered when selecting a graphics input device in a windowing environment or on a terminal, your program may not behave as expected.

The special considerations are as follows:

- If a graphics pointing device (such as an HP-HIL A-size Digitizer) is used by the windowing system, it may be accessed only as a relative locator device. Therefore, you need to use the following command to access this device and any other device used by the windowing system:

  ```
  GRAPHICS INPUT IS KBD,"KBD"
  ```

- If a relative locator device is not used by the windowing system, then BASIC/UX also cannot use that device.
- If you wish to use a tablet as an absolute locator device, it must not be used by the windowing system. If it is not used by the windowing system, you can use the following command to access it:

  ```
  GRAPHICS INPUT IS KBD,"TABLET"
  ```

- The touchscreen will not perform as expected within a windowing environment. Its use under these circumstances is not recommended.

- HPGL input devices (such as an HP 9111A Digitizer) can only be accessed by one BASIC/UX process at a time. For example, if one BASIC/UX process executes the following set of statements:

  ```
  ...
  200   GRAPHICS INPUT IS 706,"HPGL"
  210   DIGITIZE X,Y
  ...
  ```

  and the DIGITIZE statement has not been satisfied before a second BASIC/UX process executes the same set of statements, then an error will occur in the second process. It is possible for both processes to execute:

  ```
  GRAPHICS INPUT IS 706,"HPGL"
  ```

  successfully, as long as neither one tries to access the device at the same time.

- If the window pointer moves outside of the BASIC/UX root window, then BASIC/UX does not get any input until the pointer returns to the BASIC/UX window.

- Terminals (such as an HP 2393A) can only support HP-HIL input from the keyboard arrow keys. Any other HP-HIL devices attached to the terminal keyboard are not recognized by the computer running the BASIC/UX system. To use the keyboard arrow keys for graphics input, you need to execute the following command:

  ```
  GRAPHICS INPUT IS KBD,"KBD"
  ```

  Terminals can also receive input from HPGL devices (such as an HP 9111A Digitizer) connected to the computer. However, the same consideration that applies to two BASIC/UX processes running in different windows also applies to two processes running on different terminals.

Note HP BASIC/UX 700 does *not* support terminals.

Example

This example was shown in a previous section; however, it is different because the graphics input device is an HP-HIL tablet and the plotting device is the BASIC/UX root window. This program continuously tracks the input locator and monitors the pressed/not-pressed status of the HP-HIL graphics tablet's stylus. The cursor position is continuously echoed on the output device, and lines are drawn if the stylus is pressed.

```
100   GRAPHICS INPUT IS KBD,"TABLET" ! Define input device
110   PLOTTER IS 600,"WINDOW"        ! Define output device
120   GRAPHICS ON
130   VIEWPORT 10,133,15,100         ! Match aspect ratios
140   WINDOW 0,100,1,100             ! Define UDUs
150   FRAME                          ! Draw limits
160   !
170   LOOP
180     READ LOCATOR X,Y,Status$
190     SET ECHO X,Y
200     Button$=Status$[1,1]
210     GOSUB Action
220   END LOOP
230   !
240 Action:IF Button$="0" THEN MOVE X,Y
250   IF Button$="1" THEN DRAW X,Y
260   RETURN
270           !
280   END
```

Data Display and Transformations

In this section, various advanced topics are discussed briefly. After reading the discussions, load the example routines and try them out. No program listings are provided, but the programs and subprograms are on the *Manual Examples* disk (`/usr/lib/rmb/demo/manex` for BASIC/UX users). Every file has at least some code which is general enough for use in your own applications. The program files can be loaded with a LOAD command. The subprogram files are in ASCII format; they must be retrieved with a GET command. Some of the routines work on either monochromatic or color CRTs, but a few only work on a color computer. These will be noted as such.

Several external routines are called by the following subprograms. They are short, convenient utility subprograms. Note that the subprograms (including the utilities) are included for your convenience. You will need to create applications programs (PROG-type files) to use them.

Bar Charts and Pie Charts

The bar chart routine, which can plot on a CRT or a plotter, is a general purpose routine. Bar charts can be "comparative"; that is, individual bars are compared with one another. They can also be "stacked"; that is, bars from the same group are stacked one on top of the other, so that the sums of the bars in each group can be compared. (See the file `BarChart` for an example of a bar chart routine.)

The pie-chart subprogram (`Pie_Chart` on the *Manual Examples Disk*) can use both the color map and area fills. This program sends random data to the subprogram.

Two-Dimensional Transformations

When you want a two-dimensional figure drawn after having been scaled, translated, rotated, or sheared, you need to know about the generalized 2D transformations. The purpose of this manual is not to go into theoretical discussions in depth, so some excellent sources are cited at the end of the chapter.

The transformation matrices for scaling, translation, rotation, and shearing are defined as follows:

2D Scaling Transformation Matrix

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

S_x is the scaling factor in the X direction, and S_y is the scaling factor in the Y direction. This means that you can stretch or compress the image along both axes independently.

2D Translation Transformation Matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{pmatrix}$$

T_x and T_y are the translation factors in the X and Y directions, respectively. Translation (moving the image) can take place in the X and Y directions independently.

2D Rotation Transformation Matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This translation allows you to rotate the image about the origin. θ is the angular distance through which the object is rotated. If you want to rotate the object about some point other than the origin, you must translate that point to the origin, do the rotation, and translate it back to the original point.

2D Shearing Transformation Matrix

$$\begin{pmatrix} 1 & Sh_y & 0 \\ Sh_x & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Shearing is translating different parts of the image different amounts, depending on the value in the other axis. For example, if your data array is the outline of a capital "R", shearing in the X direction with a positive value would shift the top of the letter farther to the right than the middle of the letter.

These transformations are applied to the data array by a matrix multiplication (see the MAT statement in the *HP BASIC Language Reference* manual). To see these operations in action, load program `Lem2D`.

Three-Dimensional Transformations

3D Scaling Transformation Matrix

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Translation Transformation Matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

3D Rotation Transformation Matrices

Rotation about X-axis

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about Y-axis

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about Z-axis

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Shearing Transformation Matrix

$$\begin{pmatrix} 1 & S_{yx} & S_{zx} & 0 \\ S_{xy} & 1 & S_{zy} & 0 \\ S_{xz} & S_{yz} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Surface Plotting

Three different methods are included in the example programs for plotting a surface; that is, plotting a two-dimensional array the value of which elements represent the third dimension at that point. Each method displays the same data so that you can see the advantages and disadvantages of each display method. The data are random "mountains" and "valleys" and look somewhat like old hills worn smooth by erosion.

Contour Plotting

A contour map is a display of a surface from directly above the surface from an infinite distance. "Infinite" in this context means that no perspective effects are included.

The subprogram is passed the surface array, the minimum and maximum contour levels, the contour interval, and three logical variables. These specify:

1. whether or not you want the local minima and maxima noted on the output.
2. whether or not you want the two lines of information concerning array size and contour intervals.
3. whether or not the plot is to be sent to a CRT.

For more information, see the file `Contour` on the *Manual Examples Disk*. This subprogram was instructed to note the local highs and lows, and also to print the array information at the bottom of the plot.

Gray Maps

A gray map can be output to a monochrome or color CRT. On a monochrome version, the probability of a pixel being turned on is proportional to the value of the array at that point. To make computation easier, the routine scales the array such that the lowest point becomes zero, and the highest point becomes one. Therefore, the light areas are the high points, the darker areas are the low points, and the average brightness of an area on the screen is proportional to the value in the array at that point.

Also, a gray map can be drawn on a color-mapped display (not an external color monitor interfaced with an HP 98627A because the color-map capabilities are needed). Here, all pixels are turned on, and it's the *color* of each pixel which depends on the array value. (See the program `Gray_Map` on the *Manual Examples Disk*.)

Surface Plot

You can also look at an array from some angle other than straight above. The routine `Surface` (on the *Manual Examples Disk*) allows you to look at the surface from above or below. Again, since this is the same data as before, notice that the highs and lows are in the same places.

This routine functions by plotting each row of the array as one line on the plotting device. The points of each line are defined to be an offset (determined by which row is currently being plotted and the "height" from which you are looking at the surface) plus the value of the array element you are on. A height array is maintained, the first row of which is the highest point encountered thus far for that column number, and the second row contains the lowest points encountered thus far. If a point is higher than the highest point seen so far, it is visible, and then it becomes the new highest point. The low points are similarly maintained.

The parameter `Front_edge` and `Back_edge` are the height in GDUs that the front edge and the back edge of the array are to be from the bottom of the plotting surface. If `Front_edge` is less than `Back_edge`, more of the top surface will be visible. Conversely, if `Front_edge` is greater than `Back_edge`, more of the bottom surface will show.

If the variable `Opaque` is passed to the subprogram with a value of 0 (false), the surface is treated as if it were transparent, and no hidden lines are removed. This makes the surface hard to interpret because you cannot tell which surface is supposed to be closer to you; *everything* is visible. If `Opaque` is 1 (true), hidden lines are removed.

References

For an in-depth discussion into many areas of computer graphics, we recommend these books:

Principles of Interactive Computer Graphics, William M. Newman and Robert F. Sproull, 2nd Edition, McGraw-Hill, 1979.

Fundamentals of Interactive Computer Graphics, J. D. Foley and A. Van Dam, Addison-Wesley, 1982.

Mathematical Elements for Computer Graphics, David F. Rogers and J. Alan Adams, McGraw-Hill, 1976.

Seeing: Illusion, Brain, and Mind, John P. Frisby, Oxford University Press.

7

Communicating with the Operator

It is very unlikely that a computer could perform useful work without receiving input. Much of that input is from electronic devices: instruments, mass storage devices, other computers, and so on. Because a computer is an electronic device, it is very good at these tasks. There are also times when the computer's input must come from the human sitting in front of the computer.

Good human interfaces do not happen without some effort from the programmer. In many programs, at least one fourth of the code is dedicated to the human interface. It is not unusual to use one half of a good program for operator interaction, error trapping, explanatory messages, etc. Obviously, these estimates depend upon many factors, like the task being performed and the intended operators. If you are the only person who uses a program, that program may not need a friendly human interface. However, the demands for a good human interface rise greatly if a program is used by many people with different backgrounds. When the intended users do not understand computers, your program must be very skillfully written so that it does not intimidate the operator or make great demands on their ability to guess what your program wants from them.

Overview of Human I/O Mechanisms

Here are the elements of a human interface that this chapter discusses:

- Sending messages to the operator:
  - □ Displaying text for the operator to read (DISP and PRINT).
  - □ Changing display fonts (CHRX, CHRY, and SET CHR).
  - □ Generating sound (BEEP and SOUND).

- Handling messages from the operator:
  - Softkeys (ON KEY, KEY LABELS, LOAD KEY, SET KEY)
  - Rotary pulse generators (ON KNOB and ON CDIAL)
  - High-level alphanumeric input (INPUT, LINPUT, ENTER)
  - Low-level keyboard input (trapping key codes with ON KBD)

Other Factors

These are certainly *not* the only elements in a human interface. A good human interface can involve the placement of hardware, use of graphic and voice communication, data base management, artificial intelligence theories, and much more. However, you must begin somewhere. Hopefully, the hints in this chapter will help your present programs and whet your appetite for more elaborate human interfaces in future programs.

Displaying and Prompting

One of the simpler things to do for the operator is to display an explanation of what is happening or what is expected. In the early days of computers, memory was a scarce and expensive resource. Programmers were encouraged to use as little memory as possible. It seemed as though there was a contest to see who could put the most information into a 32-character message.

Please realize that those days are over. For example, there is no significant restriction on program size: the standard computer is shipped with over a half-million characters of memory, and there are usually at least 18 lines of 80 characters visible at all times on the display. If you are sending your operator tiny, cryptic messages, you are making an unnecessary mistake.

Displaying Messages: A Two-Step Process

Giving instructions to the operator can be viewed as two basic steps:

1. Prepare to use the display by putting it into a known, usable state (disable any unwanted modes, clear the screen, etc.).
2. Use as much of the display as necessary to give readable instructions.

Turning Off Unwanted Modes

There are several modes that affect the appearance of the display. Each is very useful for certain purposes; however, some are undesirable for the display of simple text. It is embarrassing to the programmer and confusing to the operator when two or more displays combine in an unplanned manner. The culprits are "left-over" alpha and "left-over" graphics. Left-over alpha can occur for a number of reasons:

- The operator may have used the knob or cursor-control keys to scroll text from the off-screen buffer.
- With TABXY, the PRINT statement overwrites any old characters on a line with new characters. However, if the old text is longer than the new text, the end of the old line remains visible. Therefore, the following statements do *not* print three blank lines. They just move the print position. Any old lines will still be on the screen.

  ```
  100  PRINT
  110  PRINT
  120  PRINT
  ```

- With color displays, each of the different display regions may have a different color alpha pen in effect.
- If the PRINTALL mode is on, all interactions on the display line and keyboard input line are sent to the PRINT/OUTPUT area; this may interact in an undesired manner with your program's display.

Most, but not all, of these modes are discussed in this section. For a complete listing of all display modes, see the "Display Interfaces" chapter of the *HP BASIC Interface Reference* manual.

Disabling and Enabling Alpha Scrolling

If you want to disable the cursor-control keys (such as ▲ and ▼) from scrolling the alpha display, execute this statement:

```
CONTROL KBD,16;1
```

This will prevent the user from interfering with which part of the alpha buffer is currently shown. This techniques is useful, for instance, to prevent scrolling graphics when using a bit-mapped alpha display (on which graphics and alpha are on the same raster).

If scrolling is currently disabled and you want to re-enable it, execute this statement:

```
CONTROL KBD,16;0
```

Disabling Printall Mode

The PRINTALL mode is canceled by writing a zero in the PRINTALL control register (1). The following statement turns off the PRINTALL mode.

```
CONTROL KBD,1;0
```

Disabling Display Functions Mode

The DISPLAY FUNCTIONS mode can make a display look sloppy. The following equivalent statements turn off the DISPLAY FUNCTIONS mode.

```
CONTROL CRT,4;0
DISPLAY FUNCTIONS OFF
```

Softkey Labels

The following statement displays the softkey labels on the bottom of the screen:

```
KEY LABELS ON
```

For most programs that use softkeys (discussed later in this chapter), this is a desirable mode to be in.

If you have an ITF keyboard, you can select which softkey menu to display:

```
SYSTEM KEYS
USER 1 KEYS
USER 2 KEYS
USER 3 KEYS
```

These menus are also discussed later in this chapter (as well as in the "Introduction to the System" chapter of the *Using HP BASIC* manual for BASIC/WS).

Clearing the Screen

You can use either of the following (equivalent) statements to clear the display screen:

```
CLEAR SCREEN
```

or

```
CLS
```

If you do not want to load the CRTX binary (with BASIC/WS), you can use an OUTPUT to the keyboard to accomplish the same purpose:

```
OUTPUT KBD;CHR$(255)&"K";
```

(The "Keyboard Interfaces" chapter of the *HP BASIC Interface Reference* manual fully describes the details of keyboard output.)

Printing Blank Lines

To print a line that is blank is a different operation from sending only an end-of-line sequence. A PRINT statement with no parameters simply sends an end-of-line sequence. If the print position is at the start of a blank line when PRINT is executed, that line remains blank. However, if there is text on that line, the text remains. This is not to say that it is "wrong" to use PRINT with no parameters. It just means that you cannot guarantee the output of a blank line by using PRINT with no parameters.

To print a blank line, blanks must be printed. One of the most convenient ways to send a line full of blanks is the TAB function. Here is a sequence that prints three blank lines:

```
100  STATUS CRT,9;Screen_width
110  PRINT TAB(Screen_width)
120  PRINT TAB(Screen_width)
130  PRINT TAB(Screen_width)
```

Disabling and Clearing Graphics Rasters

Left-over graphics can be turned off by the following statement (on displays which have separate alpha and graphics rasters):

```
GRAPHICS OFF
```

To clear the graphics raster, use this statement:

```
GCLEAR
```

Series 300/400/700 color (multi-plane) displays may be configured to use different planes for alpha and graphics so as to simulate the separate alpha and graphics rasters of some Series 200 displays.

```
SEPARATE ALPHA FROM GRAPHICS
```

Or you can also re-configure these displays so that alpha and graphics are not separate:

```
MERGE ALPHA WITH GRAPHICS
```

For further information concerning this topic, refer to the "Graphics Techniques" chapter in the *HP BASIC Programming Guide*.

Determining Screen Width and Height

The first step in displaying information on the screen is to determine its size. Programs written in this BASIC language can be used on either 50, 80 or 128-column displays. The height of displays may also vary. There are CRT status registers that contain the width and height of the screen.

If you are developing programs that will be transported between computers, status register 9 will be very helpful to you. The screen width is useful in centering displays, labeling softkeys, formatting tabular data, and other display tasks. The following statement places the screen width in a variable called `Crt_width`.

```
100  STATUS CRT,9;Crt_width
110  DISP "Screen width =";Crt_width
120  END
```

There is also a SYSTEM$ function that returns useful information about the CRT. The specifier "CRT ID" returns a string containing (among other things) the screen width and availability of highlights and graphics. The following example shows one method of determining the screen width with SYSTEM$.

```
120  Test$=SYSTEM$("CRT ID")
130  Crt_width=VAL(Test$[3,6])
140  DISP "Screen width =";Crt_width
150  END
```

You can also determine the screen's "current height," which is the number of lines currently enabled to display alpha information:

```
150  STATUS CRT,13;Height
160  DISP "Screen height =";Height
170  END
```

The number of lines returned includes key labels, system message line, keyboard input line, DISP line, and OUTPUT area. See the subsequent discussions of display regions for locations of these lines.

Changing Alpha Height

You can also change the alpha height by writing to CRT control register 13; the range is 9 lines through the maximum for your particular display (25, 26, 30, 48, or 51).

These (equivalent) statements set the alpha height to 10 lines (which yields a 3-line output area, since the 10 lines begin at the KEY LABELS area at the bottom of the screen):

```
CONTROL CRT,13; 10
```

or

```
ALPHA HEIGHT 10
```

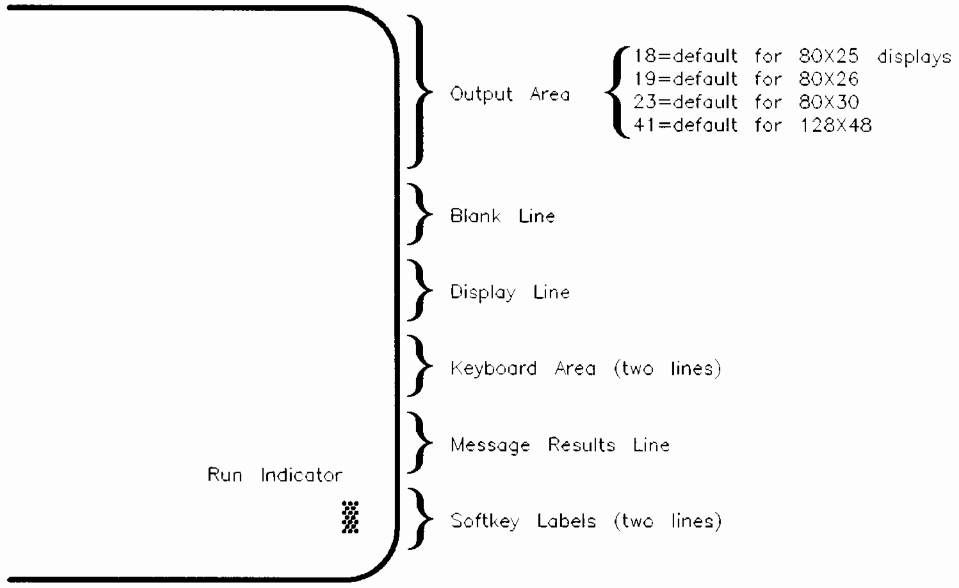
This is a handy way of specifying which part of the display is to be used for alpha and scrolling (particularly useful when using a bit-mapped alpha display and you want to use the top of the screen for graphics and the bottom part for text).

In order to return to the default alpha height, execute this statement:

```
ALPHA HEIGHT
```

Displaying Characters on the Screen

There are five regions of the display available for displaying messages for the operator.



Each requires a slightly different method of displaying characters on the CRT screen:

- PRINT and OUTPUT CRT—place characters in the "Output Area" of the screen.
- DISP—places characters on the "Display Line".
- OUTPUT KBD—places characters in the "Keyboard Input Line", just as if you had typed them in at the keyboard.

- There is no *direct* method of displaying characters in the "System Message/Results" line (but you can do it indirectly, such as with `OUTPUT KBD;"Message"&CHR$(255)&"E";`).
- ON KEY—allows you to put characters in the "Softkey Labels".

Since these topics are fully covered in the "Display Interfaces" chapter of the *HP BASIC Interface Reference* manual, they will not be discussed here. See that chapter if you are not already familiar with these keywords.

Custom Character Fonts

With displays on which the alpha and graphics share the same raster—usually called "bit-mapped alpha displays"—you can re-define the bit patterns for characters. Here are the displays that have this capability:

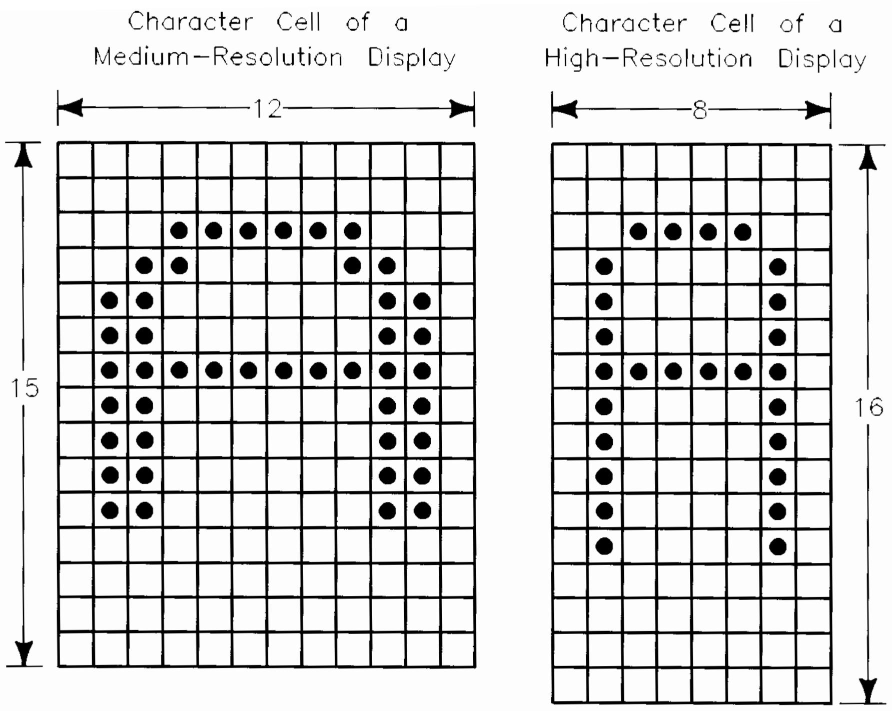
- Series 200 displays—*only* the Model 237 display.
- Series 300 displays—*all* displays *except* the HP 98546 Display Compatibility Interface.
- Series 400/700 displays—*all* displays

This display architecture makes possible the definition of custom characters (and entire fonts), with the ability to change them under program control.

Character Cells

Before getting into how to change the pixel (dot) patterns in individual characters, let's see how characters are displayed on bit-mapped alpha screens.

Display characters are produced by turning on patterns of pixels—picture elements, or dots—in the shape of the intended characters. The following diagram shows the patterns for the letter "A" for each of the two sizes of bit-mapped alpha displays:


Character Cell of a Medium-Resolution Display
12
15
Character Cell of a High-Resolution Display
8
16

Determining Character Cell Size

As shown in the above diagrams, the character cell sizes are:

- 12 × 15 for medium-resolution bit-mapped displays.
- 8 × 16 for 640 × 480 or 1024 × 768 bit-mapped displays.
- 10 × 20 for 1280 × 1024 bit-mapped displays.

You can determine the size of the character cell on the display currently in use with these BASIC functions:

- CHRY returns the height of the character cell (i.e., number of rows).
- CHRX returns the width of the character cell (i.e., number of columns).

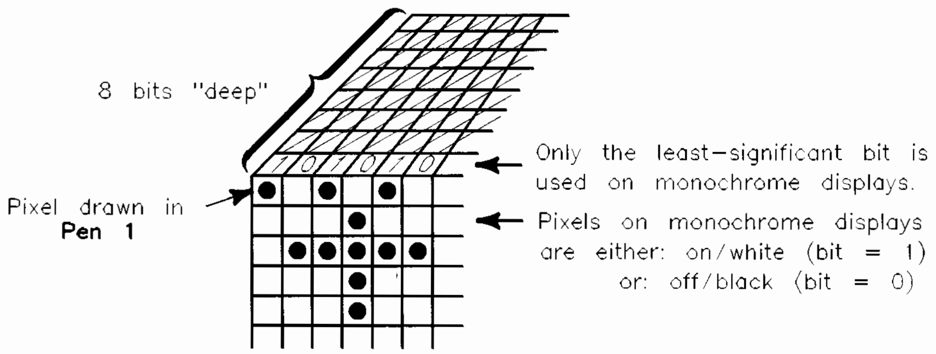
For instance, here are the results of executing these functions on a 1024 × 768 display:

```
CHRX (Return)
 8
CHRY (Return)
 16
```

Character Font Storage in Memory

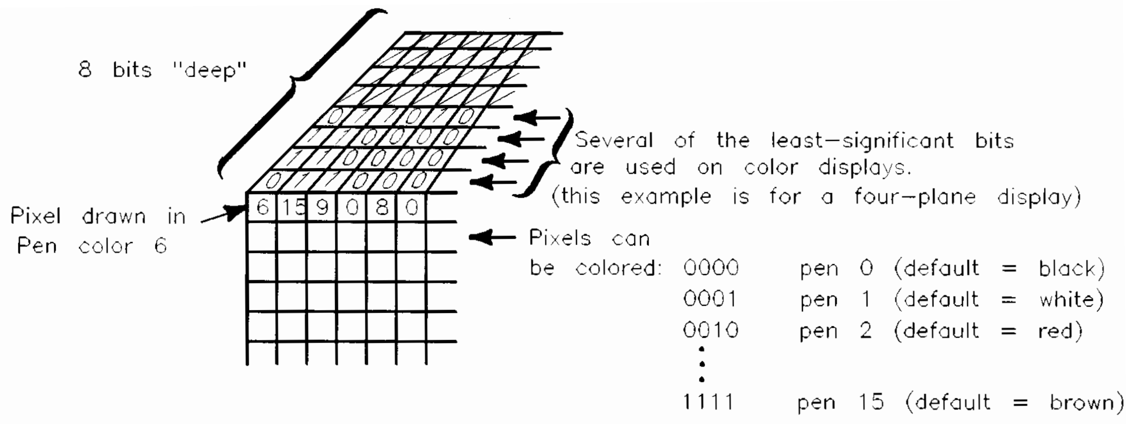
The frame buffer is an area of memory on the display card used to hold the pixel patterns shown on the screen (frame). It also has some memory which is not displayed. For each pixel of a character, there is one location in frame-buffer memory used to store the pixel.

- On monochrome displays, only the least-significant bit of this frame-buffer memory location is used (one bit "deep", "single-plane" buffer).



8 bits "deep"
1 0 1 0 1 0
Pixel drawn in Pen 1
Only the least-significant bit is used on monochrome displays.
Pixels on monochrome displays are either: on/white (bit = 1) or: off/black (bit = 0)

- On color displays, several of the least-significant bits are used—one for each plane; for instance, if it is a four-plane display (on which 16 colors can be displayed simultaneously), the low-order four bits of this INTEGER are used.


8 bits "deep"
Several of the least-significant bits are used on color displays. (this example is for a four-plane display)
Pixel drawn in Pen color 6
6 15 9 0 8 0
Pixels can be colored: 0000 pen 0 (default = black)
0001 pen 1 (default = white)
0010 pen 2 (default = red)
1111 pen 15 (default = brown)

Soft Font Usage

The font used by the BASIC system is stored in the undisplayed portion of frame-buffer memory. (These locations are read by the "Read_chrs" subprogram of the FONT_ED utility, which is explained in the "BASIC Utilities Library" chapter of the *Installing and Maintaining HP BASIC* manual. Note that the FONT_ED utility is not supported on BASIC/UX.) Whenever a character is to be displayed, the display driver (CRTB on BASIC/WS) reads the bit pattern for the character, and then writes that pattern into the display buffer (the read/write memory that is displayed on the screen).

This is the character font used by the system at the following times:

- Whenever you type characters at the keyboard.
- Whenever PRINT is executed (when PRINTER IS CRT is in effect).
- Whenever DISP is executed.
- Any time the system writes characters on the display (such as when using CAT, when the system reports an error, when softkey labels are displayed, etc.).

If you change one or more characters of this font, these characters will be used by the system in *all* of these operations which the system subsequently performs.

Restoring the Default Soft Font

The next few sections show you how to modify the font currently in memory. If you should for any reason want to restore the default font (the one in place when BASIC is booted), execute the following statement:

```
CONTROL CRT,21;0
```

This statement re-initializes the font to its boot-time default character set.

Example of Changing One Character

The following program shows an example of setting a new bit pattern for the character A for a high-resolution monochrome display, which has a 16 row by 8 column character cell (for color displays, the example should use −1 instead of 1). Note that there is an *easier* way to do this, as shown in subsequent sections; however, this example is useful to show how the soft-font re-definition mechanism (SET CHR) works.

```
100  DATA 0,0,0,0,0,0,0,0
110  DATA 0,0,0,0,0,0,0,0
120  DATA 0,0,0,0,0,0,0,0
130  DATA 0,0,0,0,0,0,0,0
140  DATA 0,0,0,0,0,0,0,0
150  DATA 0,0,0,0,0,0,0,0
160  DATA 0,0,0,1,1,0,0,0
170  DATA 0,0,1,0,0,1,0,0
180  DATA 0,1,0,0,0,0,1,0
190  DATA 0,1,1,1,1,1,1,0
200  DATA 0,1,0,0,0,0,1,0
210  DATA 0,1,0,0,0,0,1,0
220  DATA 0,1,0,0,0,0,1,0
230  DATA 0,0,0,0,0,0,0,0
240  DATA 0,0,0,0,0,0,0,0
250  DATA 0,0,0,0,0,0,0,0
260  !
270  INTEGER Char_cell(1:16,1:8)
280  READ Char_cell(*)  !  Read data above into array.
290  !
300  PRINT "Before character re-definition: ";"A A A A"
310  PRINT
320  SET CHR NUM("A"),Char_cell(*)
330  PRINT "After  character re-definition: ";"A A A A"
340  PRINT
350  END
```

Lines 100 through 250 specify the bit patterns of the new character.

Line 270 dimensions an array used to store these bit patterns (one INTEGER element per pixel.

Line 280 reads these bit patterns specified in the DATA statements.

Line 300 shows what the character looks like before it is re-defined.

Line 320 changes the pattern currently used for the character "A" to the pattern read from the DATA statements.

Line 330 prints four A's to show what the character looks like after it has been re-defined.

Editing Supplied Fonts (BASIC/WS only)

The FONT_EDitor utility on the *Utilities 1 Disk* provides a method of reading, decoding, and editing bit patterns. This section briefly describes the capabilities of this utility. For information on how to use the FONT_EDitor, see the "BASIC Utilities Library" chapter of the *Installing and Maintaining HP BASIC/WS 6.2* manual.

Font Editor Utility Capabilities

Here are the tasks we need to describe how to perform using the supplied Font Editor Utility:

- Editing bit patterns of characters in the font
- Storing the edited font (in a file)
- Loading the font into memory
- Restoring the default font (the one that was in memory at the time that the BASIC system was booted)

Re-Defining an Entire Font

The SET CHR statement was used in a preceding example to re-define one character. It can also be used to re-define an entire font. The difference is that the array that stores the bit patterns will have another dimension, which is used to index the characters in the font.

```
        .
        .
870  ALLOCATE INTEGER Entire_font(0:256,1:CHRY,1:CHRX)
        .
        .
940  SET CHR 0,Entire_font(*)
        .
        .
```

Generating Sound

Most Series 200/300/400/700 computers have the ability to generate single tones.

- On computers that are *not* equipped with an HP-HIL interface, the sound capability is limited to the BEEP statement. BEEP provides you with the ability to generate tones of software-selectable frequency and duration. See the *HP BASIC Language Reference* for the range of frequencies and durations available with this statement. For instance, the following frequency and duration generates a tone of approximately 1220 Hz for 0.25 seconds:

  ```
  BEEP 1220 , 0.25
  ```

- On computers equipped with an HP-HIL interface, there is a sound-generator chip which you can access from BASIC with the SOUND statement. You can use it in one of two modes:
  - When using simple numeric parameters (not a numeric array), SOUND allows you to generate a single tone; you may software-select *which tone generator* to use, as well as its *frequency*, *volume*, and *duration*. For instance, the following statement uses voice 1 to generate a tone at frequency 1220 Hz, of maximum volume, and with duration of 0.25 seconds:

    ```
    SOUND 1,1220,15,0.25
    ```

  - When using an INTEGER array, SOUND takes values from the array and interprets them in a special way to produce a series of tones on one or more of the available voices. Examples of this use are given subsequently in this section.

    ```
    SOUND Array_of_instrs(*)
    ```

The remainder of this section describes how to use the SOUND statement.

Example of Single Tones

Load and run the "CScale" program from the *Manual Examples Disk* (for BASIC/UX it is located in the directory **/usr/lib/rmb/demo/manex**). Here is a listing of the program. It plays all eight major notes in the key of the C.

```
120   DATA C,C#,D,D#,E,F,F#,G,G#,A,A#,B,C
130   !
140   Base_freq=523.25   !  Base_freq = C
150   FOR Note=0 TO 12
160     Freq=Base_freq*2^((Note)/12)
170     READ Note$
180     IF NOT POS(Note$,"#") THEN ! "Natural" note.
190       PRINT USING 200;Note$,Freq
200       IMAGE "Note:",X,2A,3X,"Frequency:",2X,4D.DD
210       SOUND 1,Freq,8,.5
220       WAIT .5
230     END IF ! Natural note.
240   NEXT Note
250   !
260   !
270   END
```

Here are the printed results of running the program.

```
Note:  C   Frequency:  523.25
Note:  D   Frequency:  587.33
Note:  E   Frequency:  659.26
Note:  F   Frequency:  698.46
Note:  G   Frequency:  783.99
Note:  A   Frequency:  880.00
Note:  B   Frequency:  987.77
Note:  C   Frequency: 1046.50
```

Here is a line-by-line description of the program:

Line 120 lists the notes of the scale.

Line 140 specifies the frequency of middle C. (This will be used in calculating subsequent frequencies.)

Lines 150 through 240 define a loop which reads the notes in the DATA statement and calculate the corresponding frequency (if the note is a "natural" note—not a sharp or a flat).

Lines 190 and 200 print the note and its calculated frequency.

Line 210 generates the note. (The actual frequency often does not *exactly* match the specified frequency. See SOUND in the *HP BASIC Language Reference* for a table of target frequencies and errors.)

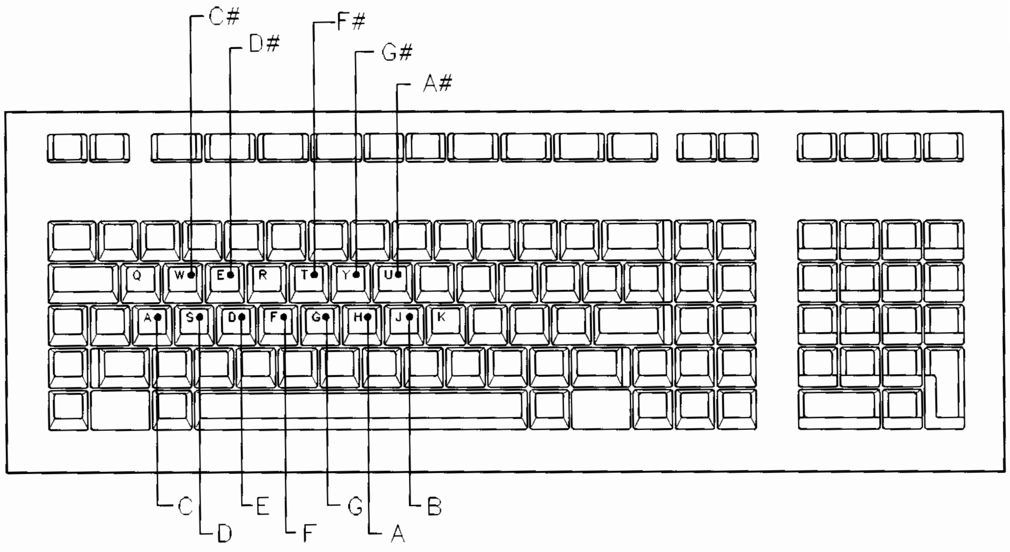
Line 220 executes a WAIT statement, which allows the note to finish playing before the next note is sent to the sound chip.

A Simple Music Editor

The "InputSong" program on the *Manual Examples Disk* (for BASIC/UX it is located in the directory `/usr/lib/rmb/demo/manex`) re-defines the keyboard to produce notes in the equal-tempered scale. The softkey menu shown by the program (when using an ITF keyboard) include: `Play`, `Load`, `Store`, and `Done`. The things you can do with the program include:

- Input a sequence of tones (see key definitions below)
- Play the notes back (press f1 or k1)
- Load notes from a file (f2 or k2)
- Store them in a file (f3 or k3)
- Quit the program (f4 or k4)

Here are the definitions of the keys provided by the program:



If you want to play an existing song, select the **Read** (f2) option and then type in the file name `OdeToJoy`. It plays a short song.

The program has been kept simple for the sake of brevity—*very* simple. It would be fairly easy to enhance the program's capabilities: allow longer songs, add some elementary editing capabilities, as well as some graphic output to the program, etc. Such modifications, to use a familiar phrase, "are left as an exercise for the reader."

Arrays of Sound Instructions

As mentioned earlier, the SOUND statement also has the ability to play several tones, based on instructions given in an array specified in the statement.

```
SOUND Array_of_instrs(*)
```

The values in the array are interpreted by the SOUND statement as follows:

Instruction	**Sound Chip Effect Produced**
0	Exit the SOUND statement (and stop reading array elements)
1 to 3	The specified voice is to be used; also says to read the *next three* array elements, and interpret them as follows, respectively: ■ tone number—used to set the frequency frequency = 83 333 / tone number ■ volume—0 = off; 1 thru 15 are lowest to highest volume. ■ duration—values 0 thru 255 are interpreted as follows: 0 is interpreted as "sound indefinitely". 1 thru 255 are interpreted as 10's of milliseconds (i.e., 1/100 second);

4	Specifies that the *noise voice* is to be used; also says to read the next *three* array elements and interpret them as above (the same as with voice numbers 1 to 3), *except* that the *tone number* parameter is interpreted as follows:
	0 => *periodic* noise; *fast* shift register clock;
	1 => *periodic* noise; *medium* shift register clock;
	2 => *periodic* noise; *slow* shift register clock;
	3 => *periodic* noise; clock shift register *with voice 3*;
	4 => *white* noise; *fast* shift register clock;
	5 => *white* noise; *medium* shift register clock;
	6 => *white* noise; *slow* shift register clock;
	7 => *white* noise; clock shift register *with voice 3*.

5 to 8	Wait for voice 1 to 4, respectively, to finish sounding before executing the next sound instruction (if any).
9	Read the following array element, and wait the specified interval (100 microseconds × that element's value) before executing the next instruction (if any).

If the end of the array is reached on one of these boundaries, then the SOUND statement terminates normally; however, if the last element of the array has been reached and the BASIC system expects to read more values, then error 17 will be reported (subscript out of range).

Executing Example SOUND Instructions

Here is a simple program that will allow you to experiment with some of these instructions. It is called "SoundInstr", and it is also on the *Manual Examples Disk* (for BASIC/UX it is located in the directory `/usr/lib/rmb/demo/manex`).

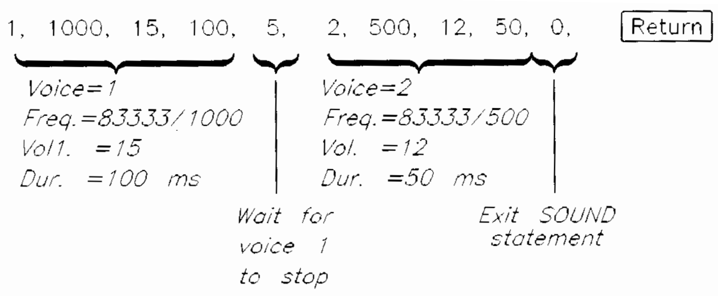
```
120   OPTION BASE 1
130   ALLOCATE INTEGER Sound_array(10)
140   !
150   DATA  1,1000,15,100,5,2,500,12,50,0
160   READ Sound_array(*)
170   LOOP
180     OUTPUT KBD;Sound_array(*), ! Put "template" on input line.
190     INPUT "Edit SOUND array parameters.",Sound_array(*)
200     !
210     SOUND Sound_array(*) ! Now execute instructions.
220     !
230   END LOOP
240   !
250   END
```

After loading the program (or typing it in), run it and begin to experiment by varying the instructions. (Use the Stop key to terminate the program.)

7

The *first time through the loop*, merely press Return to execute the instructions shown on the keyboard input line. Here are the default instructions, with an explanation of their effects.

```
Edit SOUND array parameters.
```



The *second time through the loop*, move the cursor to the left and modify one or two of the parameters, and then press (Return) to execute the instructions. For instance, this is a legal set of instructions:

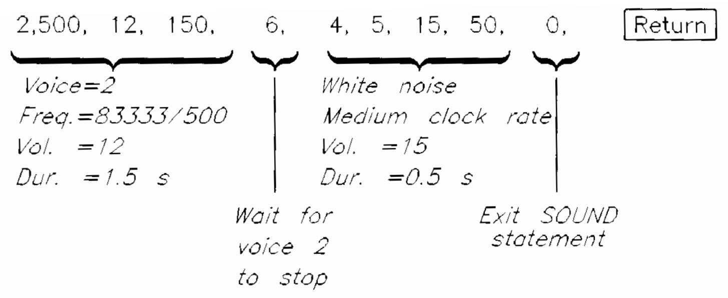
```
1, 1000,    15, 100,    6,     2,  500,  12,    50,     0, (Return)
                         |
                         |
                   Changed only
                  this parameter.
```

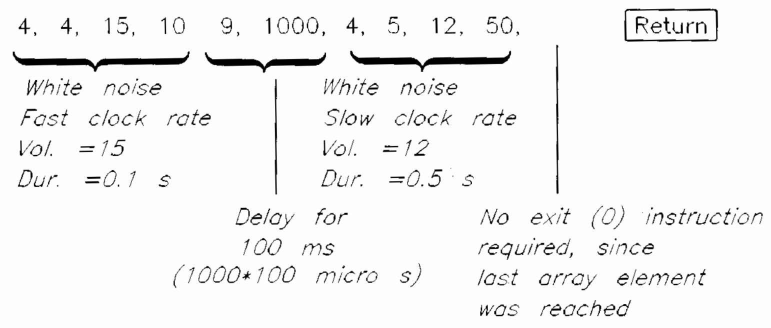
Since the only parameter that was changed was the 6, the sounds on voices 1 and 2 are now played simultaneously (instead of waiting for voice 1 to stop before starting voice 2). Note that voice 2 (the higher pitch) stops first, since it had the smaller duration parameter.

The *third and subsequent times through the loop*, you can do either of the following things:

- Press (Return) to re-play the preceding set of instructions.
- Move the cursor (◀ or ▶), modify one or more of the instructions (type over existing characters, or use (Insert char)/(Delete char)), and then press (Return) to hear the instructions' effects.

Here are several additional examples of instruction sequences and their effects.





4, 4, 15, 10 9, 1000, 4, 5, 12, 50, Return
White noise
Fast clock rate
Vol. =15
Dur. =0.1 s
White noise
Slow clock rate
Vol. =12
Dur. =0.5 s
Delay for 100 ms (1000*100 micro s)
No exit (0) instruction required, since last array element was reached

Example Song (Using SOUND Array Parameters)

The program in the file named "SoundArray" on the *Manual Examples Disk* (for BASIC/UX it is located in the directory **`/usr/lib/rmb/demo/manex`**) produces a song using the SOUND statement and an array of instructions. Load and run the program.

Operator Input

After sending messages to the operator about what you want, you can expect that you will get some sort of meaningful feedback. This section summarizes the different methods of handling operator inputs.

- Softkeys (ON KEY, KEY LABELS, LOAD KEY, SET KEY)
- Rotary pulse generators (ON KNOB and ON CDIAL)
- High-level alphanumeric input (INPUT, LINPUT, ENTER)
- Low-level keyboard input (trapping key codes with ON KBD)

Softkey Inputs

Softkeys are the keys at the top of your keyboard labeled (f1) through (f8) (on ITF keyboards) or (k0) through (k9) (on 98203 keyboards)

There are two types of uses of softkeys:

- **Typing-aids keys:** these keys generate sequences of alphanumeric and system keystrokes, which will save you time when repeatedly typing in information or commands from the keyboard
- **Program-interrupt keys:** while a program is running, the softkeys can generate interrupts (when ON KEY defines service routines for the keys)

Note that if a softkey does not have a current ON KEY definition, it can still be used as a typing-aid.

Since both of these topics are already discussed in other places of the BASIC manual set, they will not be discussed here. For further information about:

- **Typing-aid keys:** see "Introduction to the System" in the *Using HP BASIC* manual.
- **Program-interrupt keys:** see the "Program Structure and Flow" and "Interrupts and Timeouts" chapters of the *HP BASIC Programming Guide.*

However, since programmatically re-defining the softkeys is not an appropriate topic for the *Using HP BASIC* manual, it will be discussed here.

Defining Typing-Aid Softkeys Programmatically

There are two ways to programmatically re-define the typing-aid definitions of softkeys:

- Use LOAD KEY to load definitions from a file (for *all* keys). Note that LOAD KEY with no arguments restores the default definitions of the softkeys.
- Use SET KEY to load definitions from a simple string (one key) or from a string array (ranges of keys, or all keys).

The *main differences* between these statements are shown in the following table:

Method	**Source of Definitions**	**Number of Keys Defined**
LOAD KEY	BDAT file	*All* existing definitions are first cleared; then *only* keys with definitions in file are re-defined.
SET KEY	Simple string, or string array	*Single* keys or *ranges* of keys may be re-defined.

Listing Current Typing-Aid Softkey Definitions

Before getting started into how to *change* typing-aid definitions, it is handy to have a tool for checking the *current* definitions. The LIST KEY statement allows you to show these current definitions. The destination is either the current PRINTER IS device:

```
LIST KEY
```

or the specified device:

```
LIST KEY #Dev_selector
```

Storing and Loading Typing-Aids from Files

To store the current typing-aid definitions, use the STORE KEY statement. STORE KEY first creates a BDAT file of the specified name, and then stores two types of information in this file *for each key* (written in FORMAT OFF representation—for details of FORMAT OFF attribute, see the *HP BASIC Language Reference* description of ASSIGN; or see the "I/O Path Attributes" chapter of the *HP BASIC Programming Guide*).

- A key number (2-byte INTEGER)
- The corresponding typing-aid softkey's definition (a 4-byte string-length header, followed by a string of ASCII characters that comprise the key's definition)

In order to load these keys back into the computer, use the LOAD KEY statement. LOAD KEY first clears *all* current typing-aid definitions, and then loads new typing-aid softkey definitions from a BDAT file.

This file was created in one of two ways:

- By using STORE KEY (after making sure that all typing-aid softkey definitions were as desired); for example:

  ```
  STORE KEY "SOFTKEYS"
  ```

- By using OUTPUT to send the same information to a BDAT file. Here is an example program that does essentially what the preceding STORE KEY statement does (assuming the same typing-aid definitions, of course):

  ```
  100   ! File "LOAD_KEY"                ! File name of this program.
  110   DIM Key_def$[160]                ! In case of LONG definitions.
  120   INTEGER Key_number               ! 16-bit integer.
  130   CREATE BDAT "SOFTKEYS",3         ! Create a 3-record file.
  140   ASSIGN @Keys TO "SOFTKEYS"       ! Open file (default=FORMAT OFF).
  150   FOR I=1 TO 8                     ! For all softkeys (ITF keyboard).
  160     READ Key_number,Key_def$       ! Read key# and definition.
  170     OUTPUT @Keys;Key_number,Key_def$ ! Write them in file.
  180   NEXT I
  190   ASSIGN @Keys TO *
  200   LOAD KEY "SOFTKEYS"               ! Now install the definitions.
  210   !
  220   STOP
  230   DATA 5,"that",8,"work!",4,"you",7,"would"
  240   DATA 2,"I",1,"See?",3,"told",6,"this"
  250   END
  ```

  The resultant first eight softkey labels on the keyboard would be: `See? I told you that this would work!`. (Details about the number of characters available for softkey labels are shown in a subsequent section.)

The proper way for a program to handle typing-aid definitions when it does not want to make permanent modifications is to store the existing definitions in a file and reload them at exit time. Here is an example of how this can be done:

```
10 INITIALIZE ":,0",9        ! create a memory volume to hold the file
20 STORE KEY "Key_defs:,0"  ! store the key definitions in the file "Key_defs"
30 DIM A$(23)[1]
40 SET KEY 0,A$(*)          ! redefine all the keys to undefined
50 PAUSE
60 LOAD KEY "Key_defs:,0"   ! reload the old definitions of the keys
70 INITIALIZE ":,0",0       ! reclaim the memory volume storage
80 END
```

Note that LOAD BIN and memory volumes use a mark/release stack, so that the memory volume storage can only be reclaimed if:

- no LOAD BIN was done after the INITIALIZE in line 10 above
- other memory volumes INITIALIZEd after it have been reclaimed

It should be released even if the second case mentioned above is not satisfied, since a subsequent release of the later volumes will reclaim as many released memory volumes as it can.

Using SET KEY

SET KEY allows you to either define a single typing-aid or to define multiple typing-aids, depending on the string parameter you specify in the statement.

The following example program lines define typing-aid softkey f2 (k2) to clear the current line and produce some characters:

```
100  String$=CHR$(255)&"#"&"Some characters"
110  SET KEY 2,String$
```

The softkey label depends on the first few letters of the string `Some characters`. (The characters used for the label vary for different keyboards, as well as other factors. See the subsequent table for details.)

The following example program defines typing-aid softkeys f1 through f8 (k1 through k8) exactly as in the preceding example.

```
100   ! File "SET_KEY"                    ! File name of this program.
110   DIM Key_def$(1:8)[160]              ! In case of LONG definitions.
120   INTEGER Key_number                  ! 16-bit integer.
150   FOR I=1 TO 8                        ! For all softkeys (ITF keyboard).
160     READ Key_number                   ! Read key number (from DATA
statements).
170     READ Key_def$(Key_number)         ! Read corresponding key definition.
180   NEXT I
200   SET KEY 1,Key_def$(*)               ! Now install the definitions.
210   !
220   STOP
230   DATA 5,"that",8,"work!",4,"you",7,"would"
240   DATA 2,"I",1,"See?",3,"told",6,"this"
250   END
```

Here, again, the resultant softkey labels 1-8 are: `See? I told you that this would work!`.

Softkey Labels

The following table shows the number of characters available for softkey labels for each type of keyboard and display used with Series 200/300 computers.

Display Type	ITF or PC-Style Keyboards	ITF or PC-Style Keyboard in KBD CMODE	98203 Keyboard
High-resolution display (128-columns)	16 (2×8)	14 (2×7)	16
Medium-resolution display (80-columns)	16 (2×8)	14 (2×7)	14
Model 226 display (50-columns)	n/a	n/a	8

Note that the figures of "2×8" and "2×7" show that there are 2 lines of 8 or 7 characters each.

Some strings produce special effects if present at the beginning of the key label text. Most of these character sequences represent "System key" presses, such as CLR LN, (Clear line on an ITF keyboard), and STEP. You can type these characters into a typing-aid key by holding down the CTRL key while pressing the desired system key. The two-character key code produced contains a leading CHR$(255), which shows up as an inverse video k, followed by the character shown in the following table.

Characters	System Key Represented	Effect on Key Label
k S	STEP (f1)	Step displayed in key label (if these are the only 2 characters in the label).
k C	CONTINUE (f2)	Continue displayed in key label (if these are the only 2 characters in the label).
k R	RUN (f3)	RUN displayed in key label (if these are the only 2 characters in the label).
k A	PRT ALL (f4)	Print All displayed in key label (if these are the only 2 characters in the label).
k F	DISPLAY FCTNS (f6)	Display Fctns displayed in key label (if these are the only 2 characters in the label).
k $	ANY CHAR (f7)	Any Char displayed in key label (if these are the only 2 characters in the label).
k #	CLR LN (Clear line)	Not displayed in key label (if they are the 1st two characters in the label).
$^{U}{}_{L}$ CHR$(132)	"Underline" character (not a system keycode)	If the key label has two rows, and if this character is either the 1st character in the key label or immediately follows k #, the system draws a line between the top row and the bottom row of key label characters. (Otherwise, no effect on key labels.)

7

Using Knobs

The ON KNOB and GRAPHICS INPUT IS statements allow you to programmatically sense knob rotation. Knob inputs can be received from built-in knobs on 98203-type keyboards, or from HP-HIL knobs, and they can also be received from a mouse. This section only discusses how to trap knob rotation by using ON KNOB. See the "Graphics Input and Display" chapter in this manual for examples of using GRAPHICS INPUT IS.

The following program is a very short example that demonstrates a real-time interaction between the knob and the CRT. If you run this example program and turn the knob, you will see the kind of interaction that might be used for cursor control in a text editor, for instance. Obviously, a real cursor-control routine would be much more sophisticated, but this demonstrates the basic idea.

```
100  ON KNOB .1 GOSUB Move_blip
105  STATUS CRT,13;Alpha_height
110 Spin:  GOTO Spin
120  !
130 Move_blip:  !
140    PRINT TABXY(Spotx,Spoty);" ";  ! Erase old 'blip'.
150    Spotx=Spotx+KNOBX/5            ! Scale knob inputs.
160    Spoty=Spoty+KNOBY/5
170    IF Spoty<1 THEN Spoty=1        ! Check range.
180    IF Spoty>Alpha_height THEN Spoty=Alpha_height
190    IF Spotx<1 THEN Spotx=1
200    IF Spotx>50 THEN Spotx=50
210    PRINT TABXY(Spotx,Spoty);CHR$(127); ! Display new 'blip'.
220    RETURN
230  END
```

This example uses a short infinite loop to wait for pulses from the knob (line 110). Interrupts from the knob are enabled by the ON KNOB statement in line 100. The service routine erases the old "blip", performs some scaling and range checking on the knob input, and prints the new "blip".

The scaling and range checking are very important in this kind of interactive routine. Humans expect their interface to have a certain "feel." Displays should not change too quickly or too slowly. Certain kinds of displays are expected to change logarithmically, others are expected to change linearly. The boundary values of variables are expected to conform to the boundaries of the display. To initiate yourself to some of these concepts, try modifying this simple example. Remove one or more of the range checking lines. (An easy way to do this kind of editing is to place an exclamation point in front of the statement. This turns it into a comment, removing it from the flow of execution. But it can be easily returned to the program by deleting the exclamation point.) Also try changing the scaling factor in lines 150 and 160. Notice the "feel" that results from larger and smaller increments, or even logarithmic scaling.

Using Control Dials

BASIC provides the ability to set up event-initiated branches upon detecting the rotation of knobs on "Control Dial" devices (such as the HP 46085A Control Dial Box).

Keywords and Capabilities

There are three BASIC keywords for accessing multiple-knob devices:

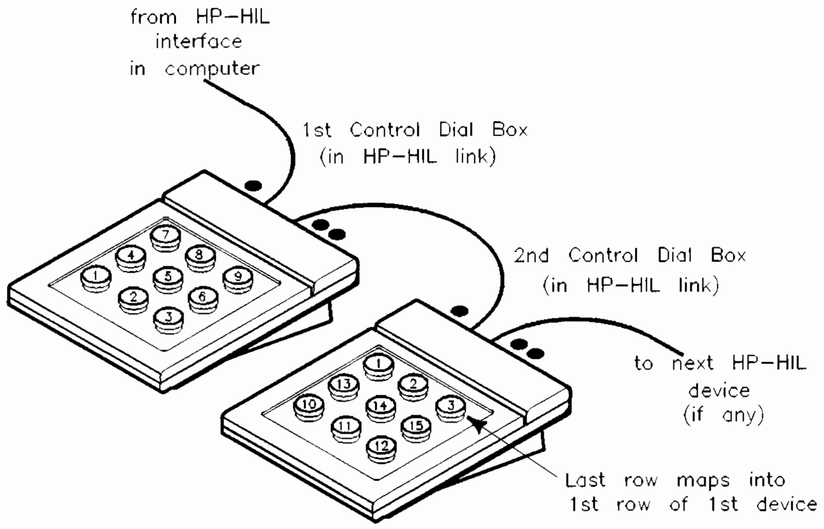
- ON CDIAL—sets up and enables interrupt branch upon detecting rotation of one of the control dials.
- CDIAL—interrogates the BASIC system to determine:
  - Which knob(s) have been rotated?

`CDIAL(0)`	returns a 16-bit status word, with each bit corresponding to a dial number (for example, bit 15 set indicates that dial number 15 has been rotated, while bit 1 indicates the same for dial 1.

  - How much a particular knob has been rotated?

`CDIAL(1)`	returns the number of pulses accumulated for dial 1;
`CDIAL(2)`	returns the number of pulses accumulated for dial 2;
.	
.	
`CDIAL(15)`	returns the number of pulses accumulated for dial 15.

Here is the numbering of dials used by CDIAL:


from HP-HIL interface in computer
1st Control Dial Box (in HP-HIL link)
2nd Control Dial Box (in HP-HIL link)
to next HP-HIL device (if any)
Last row maps into 1st row of 1st device
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

- OFF CDIAL—disables interrupt branching for control dials.

Does BASIC/UX See the Control Dial Box?

You can execute the program titled "HIL_ID" in **/usr/lib/rmb/demo/manex** to verify which Human Interface Link (HP-HIL) devices are accessible by BASIC/UX.

Some HIL devices, such as the keyboard and mouse, in the X Windows environment are global resources and will not be recognized by the HIL_ID program.

The "Verifying and Labeling Devices" chapter of the *Installing and Maintaining HP BASIC* manual describes how to check to see that HP Human Interface Link (HP-HIL) devices have been properly connected to the computer, are functioning correctly, and have been logged in by the BASIC system. If you have not performed that verification yet, you should do so now.

An Example Control Dial Handler

The following example program sets up an interrupt service routine for one Control Dial box. The program is named "CDials", and it is on the *Manual Examples* disk or directory. The program draws a box in a 3-dimensional coordinate system. The dials are defined to perform the following actions:

Dial 1	Changes the "X" location of the box.
Dial 2	Changes the "Y" location of the box.
Dial 3	Changes the "Z" location of the box.
Dial 4	Changes the "X" size of the box.
Dial 5	Changes the "Y" size of the box.
Dial 6 thru 9	No action has been implemented.

Here is the pertinent part of the interrupt-service routine for the Control Dial box:

```
350   Bits=CDIAL(0)          ! Read 16-bit status word (which knobs?)
360   !
370   IF BIT(Bits,1) THEN X=X+.1*CDIAL(1) ! Dial 1 turned; change X pos.
380   IF BIT(Bits,2) THEN Y=Y+.1*CDIAL(2) !  "   2 turned;    "    Y pos.
390   IF BIT(Bits,3) THEN Z=Z+.1*CDIAL(3) !  "   3 turned;    "    Z pos.
400   !
410   IF BIT(Bits,4) THEN X_size=X_size+.2*CDIAL(4)  ! Change "X_size".
420   IF BIT(Bits,5) THEN Y_size=Y_size+.3*CDIAL(5)  ! Change "Y_size".
430   !
```

Line 350 interrogates the "status word" to determine which dial(s) have been rotated. For each one that has been rotated, the corresponding action is taken (lines 370 through 420). For instance, rotating dial 1 moves the box along the X axis (while the Y and Z coordinates remain fixed). Rotating dial 5 changes the "Y size" of the box (while the "X size" remains constant).

In order to implement additional functions, all you need to do is to add similar IF ... THEN statements (or segments) that execute the appropriate action.

Accepting Alphanumeric Input

When possible, it is a very good choice to used only softkeys and knobs to get input from the operator. It eliminates the need for translating an endless variety of typing mistakes that might be supplied as input to program variables. Softkey input is very tightly controlled by the programmer. Unfortunately, it is often necessary to leave that comfortable, controlled world. Suppose you need to get a device selector from the operator. You can't very well define a softkey that increments a variable and expect the operator to press it 701 times!

The proper handling of keyboard input may be one of the most neglected areas of applications programs. Programmers often fail to see the program as users see it, underestimate the potential for operator error, and balk at the amount of code needed to skillfully handle incoming text. However, you need not write input routines that can parse broken English with misspelled words. The objective is simply to keep the program from terminating and to take some unnecessary pressure off the operator. Obviously, a program can't tell if the operator misspelled a file name until it accesses the disk. Therefore, error trapping is an important part of handling operator input.

One task that can be performed by the input routine is *anticipating common problems*. Many of these are covered in this section's examples, but here is a preview. You know that exceeding the dimensioned length of a string gives error 18. So don't use short strings in an INPUT statement. You know that CAPS LOCK might be on or off when the operator starts typing. So use an upper-case function to compare input with constants. You know that an operator is likely to just press (CONTINUE) if he isn't sure how to respond. So use reasonable defaults and don't try to send a null string to a NUM function.

Get Past the First Trap

Before you can do anything with a keyboard input, the computer must satisfy the items in the input list and complete the input statement. There are two keywords available for accepting input from the keyboard line: INPUT and LINPUT. Let's start by looking at the features of these two statements.

The main advantages of INPUT are:

- Either numeric or string values can be input.

- If a variable does not receive a value from the keyboard, the value of that variable is left unchanged.
- A single INPUT statement can process multiple fields, and those fields can be a mix of string and numeric data.

The INPUT statement can be powerful and flexible. When you know the skill level of the person running the program, INPUT can save some programming effort. However, this statement does demand that the operator enter the requested fields properly. To find out the details of INPUT, see the *HP BASIC Language Reference*. This section discusses an alternative to INPUT that can make fewer demands on the operator. Some of the *disadvantages* of INPUT are:

- Improper entries to numeric variables can cause errors such as "string is not a valid number" and overflows.
- Certain characters can cause problems. Commas and quote marks have special meanings and are the primary offenders.
- If DISP is used to supply a prompt, and multiple values are entered separately, the prompt is lost.

The problem with INPUT is that the program is powerless to overcome the disadvantages. If you are asking for a numeric quantity, and the operator keeps trying to enter a name, the program will never leave the INPUT statement. The operating system will beep and display error 32 until the operator gets tired or gets smart. In the event of an error, the computer automatically re-executes the INPUT statement until the operator satisfies all the requirements. Your program never gets a look at his input and you can't trap the errors.

The LINPUT statement can help with these potential problems. LINPUT stands for "Literal INPUT." The result of any LINPUT statement is a single string that contains an exact image of what the operator typed. If (CONTINUE) ((f2) on the ITF keyboard) is pressed with no entry, the result is the null string. (Nothing typed, nothing returned.) If you need things like default values, numeric quantities, and multiple values, you will need to process the string after you get it.

Since LINPUT accepts any characters without any special considerations, the only normal error would be string overflow. If the string used to hold the

LINPUT characters is dimensioned to 256 characters or more, it becomes impossible to overflow the string from the keyboard line. Therefore, LINPUT is a very "safe" way to get data from the keyboard line. The following example shows some common techniques for accepting operator input.

Entering a Single Item

This program segment requests the current month for use later in the program. A detailed discussion follows the listing. Note that the general techniques presented can be used to process many kinds of input. Entering a month is merely a convenient example.

```
100   OPTION BASE 1
110   DIM In$[160],Months$(12)[3]
120   INTEGER Temp,Current_month
130   OUTPUT KBD;"SCRATCH KEY ⓚE"; ! Typing aids distracting if not needed
140   FOR Temp=1 TO 12
150     READ Months$(Temp)            ! String data for month names
160   NEXT Temp
170   DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
180   Current_month=3                 ! Default value
190   !
200 Try_numeric:   !
210   DISP "Enter the month.  Default = ";Months$(Current_month);
220   LINPUT "",In$                   ! Ask for operator input
230   IF NOT LEN(In$) THEN            ! Check for no input
240     Temp=Current_month            ! Use default value
250     GOTO Found
260   END IF
270   ON ERROR GOTO String            ! If no numerals, may be a string name
280   ENTER In$;Temp                  ! Try to extract a number
290   OFF ERROR                       ! ENTER worked; change error trap
300   IF Temp<1 OR Temp>12 THEN Not_valid   ! Check for impossible month
value
310   GOTO Found                      ! Value is OK; use it
320   !
330 String:  !
340   OFF ERROR                       ! ENTER error trap no longer needed
350   In$=UPC$(In$)
360   FOR Temp=1 TO 12                ! Search for 1st three letters of
month
370     IF POS(In$,Months$(Temp)) THEN Found  ! Match found; use that value
380   NEXT Temp                       ! If loop finishes, no match was
```

```
found
390   !
400 Not_valid:   !
410   BEEP
420   DISP "Not a valid month. Please try again."
430   WAIT 2
440   GOTO Try_numeric
450   !
460 Found:   !
470   Current_month=Temp
480   !
490   ! Program execution continues here
```

The first statement after the variable declarations removes the typing-aid key definitions. This is done with an OUTPUT to the keyboard because SCRATCH commands cannot be stored as a program line. You may or may not want to include this in your programs. If you are not using softkeys, the presence of softkey labels may be distracting to the operator. They may indicate that many response choices are available when the keys are actually unrelated to the current question. On the other hand, your program may have loaded the typing aids with responses intended to help the operator. This is possible, but was not done in the example. Obviously, if KBD is not present, the SCRATCH KEY command will generate an error and shouldn't be included. For another method of removing the typing-aid key definitions, read the section in this chapter titled "Storing and Loading Typing-Aids from Files."

An interesting feature of this example is that the operator may respond with the number of the month, the name of the month, or an abbreviation of the name of the month. The array `Months$` is loaded with the first three letters of each month name so that name responses can be identified.

The final initialization step is to provide a default for the current month. When possible, requests for input should be accompanied by a default. If the default is well chosen, this increases the chances that the operator will not have to do any typing. Even if the default will usually be changed, it can help show the operator an acceptable format for the response.

The prompts available with INPUT and LINPUT statement must be literals and therefore cannot show any program variables. This restriction is easily overcome. Prompts appear in the same line as DISP items. The DISP statement can contain variables. To use DISP items as a prompt, a trailing semicolon is used in the DISP statements, and a null prompt is used in the LINPUT statement. This is a very useful technique that is applicable to both LINPUT and single-prompt INPUT statements.

After the keyboard input is received, the first check determines if any data was entered. It is reasonable to assume that the space bar might have been bumped accidentally during any keyboard input. The TRIM$ function corrects this "problem." A null input indicates that the operator wanted the default value, so no further processing is done.

The next check is to see if the number of the month was entered. Numerals can be converted to numeric data with the VAL function, but this demands the same strict format as INPUT. A much more powerful and flexible way to extract numeric data from a string is by using the ENTER statement. Admittedly, it is not likely that an operator would enter extra text with the number—but why generate an error if he does? The LINPUT/ENTER combination can extract the month from responses like these:

```
4
"4"
MONTH=4
4th month
```

If a number is found, the error trap is disabled. In actual applications, the OFF ERROR statement would be replaced by an ON ERROR statement that re-establishes the normal error trapping used in the program. The final check ensures that the month is within a meaningful range. You want to give the operator maximum flexibility, but accepting the 54th month is *too* flexible. Range checking is a technique that should be used in all good operator interfaces.

Although ENTER can do a lot, it cannot extract a number from a string that has no numerals. Since the operator is permitted (and encouraged) to use the name of the month, the program must handle this case. That is the purpose of the ON ERROR statement before the ENTER. If the ENTER cannot find any numeric value, the error trap directs program execution to the segment labeled `String`. This segment changes the error trap, since it has served its purpose. Then the input data is searched for the presence of a month name. A string comparison could be used, but that requires that the month name be in a fixed location within the response. Again, there is no reason for such a restriction. The POS function will find the desired letters anywhere in the line. The UPC$

function eliminates any requirements about letter case. Thus, responses like the following would all be valid:

```
JAN
January
MONTH=JAN
"january"
```

In any keyboard-input situation, there is always some possibility that the operator entered pure garbage. If all the attempts to find a meaningful number or name fail, an error message is displayed, and the entire process is repeated. Another programming choice is to assume the default if no meaningful input is found. You must judge for yourself which choice is best. If accurate operator input is very important to the program, then the program should keep asking until the operator gets smart. If the value in question is not important, it might be best the assume a default and move on to the next stage of the program.

Note that the desired variable, `Current_month`, is not updated unless a valid input was received. All the testing and searching is done using a temporary variable. This is done so that the default value is not destroyed by an invalid input.

LINPUT with Multiple Fields

This example requests the entire date: day, month, and year. As in the previous example, there is nothing special about dates. The techniques shown have general applications. A detailed discussion follows the listing.

7

```
100   OPTION BASE 1
110   DIM In$[160],Months$(12)[3],Left$[2]
120   INTEGER Temp,Current_day,Current_month,Current_year
130 Fmt:  IMAGE #,2D,",",3A,",",K,K      ! Format of date input
140   FOR Temp=1 TO 12
150     READ Months$(Temp)               ! String data for month names
160   NEXT Temp
170   DATA JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
180   Left$=CHR$(255)&CHR$(72)           ! Moves cursor to beginning of
line
190   Current_day=1                      ! Set up default values...
200   Current_month=11                   ! In real applications, these might
210   Current_year=1982                  ! come from the clock or a file.
```

```
220   !
230 Get_date:   !
240   OUTPUT KBD USING Fmt;Current_day,Months$(Current_month),
Current_year,Left$
250   LINPUT "Enter the date, using this format.",In$
260   ON ERROR GOTO Not_valid            ! No numerals = error for ENTER
270   ENTER In$;Temp                     ! Extract the day
280   OFF ERROR                          ! ENTER worked; change error trap
290   IF Temp<1 OR Temp>31 THEN Not_valid  ! Check for impossible
day-of-month
300   Current_day=Temp                   ! Value OK; use it
310   !
320   Temp=POS(In$,",")                  ! Look for first delimiter
330   IF NOT Temp THEN Not_valid         ! No delimiter = bad format
340   In$=UPC$(In$[Temp+1])              ! Remove date field; make upper-case
350   FOR Temp=1 TO 12                   ! Try to find 1st three letters
360     IF POS(In$,Months$(Temp)) THEN Found_month
370   NEXT Temp
380   !
390 Not_valid:   !
400   OFF ERROR                          ! Change ENTER error trapping
410   BEEP
420   DISP "Improper entry. Please try again."
430   WAIT 2
440   GOTO Get_date                      ! Start over with this routine
450   !
460 Found_month:   !
470 Current_month=Temp
480 ON ERROR GOTO Not_valid
490   ENTER In$;Temp                     ! Extract the year
500   OFF ERROR                          ! ENTER worked; change error trap
510   IF Temp<100 THEN Temp=Temp+1900    ! Maybe there is no century?
520   Current_year=Temp                  ! Value OK; use it
530   !
540   ! Program execution continues here
```

The first segment declares the variables, stores the month abbreviations, establishes some defaults, and contains an IMAGE statement that specifies the desired date format. Although defaults are important, program constants are not always the best way to supply defaults. Using the constant “12” as a default for a GPIO interface select code makes sense. But the date will almost always be different from a constant stored in the program. A real program should adopt some other method of assuming the date. If your computer has a battery-backed real-time clock, the date might be extracted from the clock value. If the program uses a file with the date stored in it, the last access date might be close to the current date.

A significant feature of this example is the handling of multiple fields. Multiple fields bring with them two special considerations. First, there is the need to show the operator the proper format for the fields. Second, there is the need to extract those fields from a single string, assuming that LINPUT is used.

The proper format for the fields is shown to the operator by using an OUTPUT to the keyboard. The default values are sent to the keyboard line, formatted by an IMAGE statement. This not only gives the operator the choice of simply pressing CONTINUE (f2 on an ITF keyboard), but it also shows the appearance of a correct response. If the default date is generated by a good source, it is reasonable to expect that the "day" field will be changed more often than the month or year. Therefore, the OUTPUT to the keyboard finishes by placing the cursor at the beginning of the line, in the day field.

The ON ERROR/ENTER technique is similar to the previous example. The ENTER statement extracts only the day because the comma terminates that field. The day is checked against reasonable limits and assigned to the actual variable if it is acceptable. This range checking could be expanded to check for the maximum day allowed in a specific month.

After the day is extracted, the string is searched for the comma delimiter, and the day field is removed. This is done to prevent the day number from interfering with the extraction of the year number. The resulting string is searched for the month name using the same technique as the previous example.

The year is extracted using the ENTER technique. If a valid number is found, one last test is performed. The response might have contained only the last two digits of the year. This is not likely, since the recommended format showed all four digits; but why complain if it happens? If only two digits are found, the program supplies the 1900 automatically. By the way, this technique is not too effective if the dates being entered might cross century boundaries.

Yes and No Questions

Frequently, all the computer needs from the operator is a simple "yes" or "no." The "Expanded Softkey Menu" example showed one way to handle yes/no states. However, that much processing is not always desired. If you only need to ask a single question, why program 10 softkeys and 18 CRT lines? The following user-defined function shows some simple, but friendly, processing for yes/no answers.

The objective of this routine is to provide as much flexibility as possible. This means that we don't bother the operator about such things as bumping the space bar, pressing CAPS LOCK (Caps on the ITF keyboard), or responding with a simple CONTINUE (f2 on the ITF keyboard). The main program provides a prompt or explanation and performs a LINPUT with a 256-character string. It then passes that string to this function and tests the results.

The function uses a local copy of the string just in case you need the actual input for some other purpose in the main program. The response is trimmed and placed in upper-case. Then the first letter is tested. Four cases are identified: the answer was "Y" (for yes), the answer was "N" (for no), no answer was given, or the answer was not recognized.

```
2000  DEF FNYes(X$)
2010    DIM Temp$[1]
2020    Temp$[1,1]=TRIM$(X$)
2030    SELECT Temp$
2040    CASE "Y","y"
2050      RETURN 1
2060    CASE "N","n"
2070      RETURN 0
2080    CASE " "
2090      RETURN -1
2100    CASE ELSE
2110      RETURN -2
2120    END SELECT
2130  FNEND
```

As mentioned previously, every question should have a default answer. The default answer for a yes/no question depends greatly upon the nature of the question. If you are asking the operator for permission to use standard, reasonable parameters for an operation, then "yes" is a helpful default. If you are asking for permission to initialize a disk and destroy all files, then the default answer had better be "NO"! When a question or choice occurs more than once in a program, it is usually a good technique to use the operator's previous response as the default. Put yourself in the user's place and think about how the program should run.

To use this function to best advantage, the result must be tested thoughtfully. If the operator simply presses (CONTINUE) ((f2) on the ITF keyboard), the result will be −1. Therefore, the default should be assumed if `FNYes=-1`. A "yes" answer is indicated by `FNYes=1`; whereas a *non-negative* answer can be tested simply as `IF FNYes`. A *non-affirmative* answer is `FNYes<1`. Any result less than zero is a noncommittal reply. Perhaps the default could be assumed for any negative result, or perhaps a negative result should cause the question to be repeated. The test `IF NOT FNYes` reveals a negative reply. As you can see, many shades of interpretation are possible.

Example Human Interfaces

This section puts together some of the techniques discussed in preceding sections.

An Expanded Softkey Menu

A good human interface often involves the coordination of multiple resources. The softkeys are a very good tool for accepting operator input. The biggest problem with using softkeys is the severe limitation on the number of prompt characters associated with each key. Therefore, a softkey interface is an appropriate task to demonstrate the increased use of CRT space.

The goal of this technique is to display a readable and informative menu that monitors the operator's input. The example program `Softkeys` on the *Manual Examples Disk* displays a summary of the parameters that are controlled by softkeys. This summary is updated every time a softkey is pressed, providing immediate feedback to the operator. This example uses many of the CRT-control techniques already presented. It also helps to show why the human interface of a program can require so much code. This program segment simply logs the operator's choice of four items, and it is over 100 lines long. The purpose of each section of code is explained below.

The program uses softkeys 5 through 9. If you have an ITF keyboard, your softkeys are labeled 1 through 8. You can modify the program to use the softkeys most useful for your applications.

It is always good programming practice to declare all variables. The first two lines do this. Next, the variables are given their starting values. Initialization is completed by turning off unwanted modes and clearing the CRT.

The section at "Menu" displays a description and current status for each menu item. This example shows some of the parameters that might be used by a simple text-printing program. The items used are representative only. A real text formatter would have many more parameters (all the more reason to present them clearly). The operator can choose the following:

- Back-slash or up-caret as a command delimiter
- Right or left disk drive for the source of the text
- Standard or alternate format for the text
- Page numbering with Arabic or Roman numerals

Notice some important aspects of this menu. All items have default values and all defaults are visible simultaneously. This is very important. It is irritating and confusing when an operator must answer question after question to get a program to begin. It is far better to show the default environment and allow a single keypress to start the program if the defaults are acceptable. If any defaults need to be changed, the operator changes only those items he wants to change. He can press "START" at any time, and in this simple case, never answers any questions. The operator wants a printout, not a game of "20 questions."

The current state of all items is displayed in a form that is meaningful to the operator. It is reasonably safe to assume that all operators know what "RIGHT" and "LEFT" mean. Very few would have any idea what ":INTERNAL,4,1" means. Programmers need to learn about concepts like "mass storage unit specifier." Operators shouldn't be bothered by such things. Likewise, don't expect anyone to answer "1" or "0" to a question that should be answered "YES" or "NO."

A more technical aspect of this menu is the method used to update the display. Since the scrolling keys are on one side of the softkeys and the knob is on the other side (of 98203 keyboards), it is reasonable to assume that the operator might accidentally move the display out of place. One way to correct this would be to start each display update with a CLEAR SCREEN statement. This guarantees the state of the CRT and the print position. Unfortunately, it also causes a very undesirable "blinking off" of the display each time a key is pressed. A constantly disappearing menu is very distracting. The BASIC system now has the capability of disabling scrolling. See the discussion near the beginning of this chapter for details.

The objective is to give the impression that nothing changed except the selected item. Therefore, the "clear" sequence is sent before the first display only. Subsequent updates use a "home" sequence to ensure the position of the text, and a TABXY to set the print position. As a result, the new menu is written on top of the old menu. (The same visual effect could be achieved by using individual TABXY functions to access each item display, but that is a more difficult program to write.)

Since the old display is overwritten each time, it is important to erase all unneeded characters. Notice that the "NO" displays are padded with a trailing blank to erase the "S" left over from "YES." This technique can be extended to clear old displays of unknown length. The following example displays a number and erases any remaining digits from the old number. The variable `Screen` contains the screen width.

```
1300  PRINT Value;TAB(Screen)
```

The example also uses screen width for centering. Centering is not as important as keeping the display properly updated, and centering slows down the update process slightly. However, the technique is shown here in case you want to use it. During the initialization of variables, the current screen width is determined. This might be 50, 80, or 128 characters if the program is used on different models of computers. The width of the menu display is subtracted from the screen width to determine the amount of left-over space. If half of this space is sent at the beginning of the line, the remaining half will be at the end of the line. This produces a centered display. The amount to be sent at the beginning of the line is placed in the variable `Center`. This value is used to position the start of each line and is also used as a reference point to position the second column.

Models with ITF keyboards allow 16 characters (2 rows of 8) in a softkey label. Models with 80-column CRTs allow 14 characters in a softkey label. Models with 50-column CRTs allow only 8 characters for these labels. Therefore, the variable `Screen` is also used to control the display of softkey labels. This is the purpose of the segment at line 1440. The alternative is to restrict all softkey labels to 8 characters. This is possible, but undesirable. It is difficult to say anything meaningful in 8 characters. Users with 80-column CRTs will appreciate the extra meaning that is available with longer labels. The 128-column CRT can use longer labels, but this program uses the 14-character labels.

The ON KEY statements for keys 0 through 4 are used to turn off any typing-aid definitions that might exist for those keys. An ON KEY definition overrides a typing-aid definition when the program is running. However, if no ON KEY definition is supplied, the typing-aid definition remains active. This is not desirable when you are trying to achieve a program-controlled softkey menu. Therefore, the unused keys are given a "dummy" ON KEY definition to keep the menu clean. For ITF keyboards, you should "turn off" all 24 softkeys.

Notice also that when five or less softkeys are used, keys 5 through 9 are defined. This is to accommodate the Model 216 small keyboard. On the small keyboard, those are the unshifted keys. Why make the operator press the shift key? If you have an ITF keyboard, use keys 1 through 5.

The softkeys are defined to send program execution to a parameter-changing routine. Each such routine ends by sending program execution to the display-update routine. In this example, there is no demonstrated reason for repeating the ON KEY definitions for every keypress. Those definitions could have been placed above the "Menu" line and executed only once. However, some applications might need to change the key definitions in response to changes in program variables. For example, a key that produces an "insert" operation would be disabled when enough inserts had been performed to fill an array. Also, it is possible to include the value of a string variable in a key label. Therefore, the key labels may need to be rewritten as new selections are made. In cases like these, the ON KEY statements need to be in the update path.

Note This example is intended for use on an HP 98203 keyboard. For an ITF keyboard, the default conditions for key labels are slightly different. In this case, you can use KEY LABELS OFF to turn the softkey labels off, and KEY LABELS ON to re-enable displaying them. You can alternatively use CRT control register 12 to set the key-labels display mode to match the behavior of the 98203 keyboard. See the *HP BASIC Language Reference* description of the KEY LABELS statement for details.

The final "cleanup" action takes place when the operator presses "START." This is the signal that the selection menu is no longer needed. The menu display is cleared to reflect the fact that it is no longer in use. The OFF KEY statement performs two functions. It turns off the softkey label area, which helps keep the CRT neat. More importantly, it cancels all the ON KEY branches. If this were not done, the operator could cause the program to jump back to the selection menu at any time. This is probably not desirable. You may want to define some sort of "Abort" key that lets the operator stop a lengthy operation. But it is not likely that the selection menu would be the destination of an abort operation. Remember, ON KEY definitions stay around forever unless you turn them off or the program stops.

Not much has been said about the parameter-changing routines. The examples shown use a simple IF ... THEN ... ELSE structure to select between to alternatives. This concept can be expanded to allow selection of more than two choices. The MOD function is handy when you want to cycle through several choices. The following example shows a routine that rotates through four choices. This routine is intended to fit into our menu selection process. Accent protocols for different languages are shown here, but the technique is applicable to any selection item.

```
1910 Accents:  !
1920  Lang=(Lang+1) MOD 4          ! Choose accent protocol
1930  SELECT Lang
1940  CASE 0
1950    Language$="ENGLISH"
1960  CASE 1
1970    Language$="FRENCH "
1980  CASE 2
1990    Language$="SPANISH"
2000  CASE 3
2010    Language$="GERMAN "
2020  END SELECT
2030  GOTO Menu
```

Moving a Pointer

Many programs have a main menu from which the operator chooses a subtask. An example might be an editing program that gives the choice of getting a file, storing a file, editing a file, merging files, listing a file, protecting a file, deleting a file, etc. As with all other tasks, there are many ways to present this choice to the operator. Each task might be assigned to a softkey. The ON KBD statement might be used to equate individual keys to each task. For example, E for edit, M for merge, G for get, and so on. Depending on the application, one of these methods may be good. However, there are some considerations. There might be more choices than softkeys, or the arrangement of the softkeys might be awkward. The single-letter method is always just a little "dangerous". What if the operator tries to type a word? Did "P" stand for "protect" or "purge"?

One alternative is to display all the choices, with a pointer to the current selection. When the operator is sure that the selection is proper, a single press of a softkey tells the computer "Do it." The menu choices can be full phrases with no abbreviations, since the whole CRT is available for the display. The pointer can be moved by softkeys or by the knob. Since we just discussed the softkeys, let's use the knob for this example.

The following example clears the CRT, displays seven selections, and allows the knob to cycle a pointer through the selections in either direction. In a real application, meaningful phrases would be used to identify the selections, and a softkey would be defined to start the selected process. Softkeys could also be used to move the pointer up and down. This could be in addition to the knob or in place of it. A detailed discussion follows the listing.

```
100   DIM Marker$[4],Home$[2],Clear$[2]
110   INTEGER Point
120   !
130   Clear$=CHR$(255)&CHR$(75)       ! CLEAR SCR key
140   Home$=CHR$(255)&CHR$(84)        ! HOME key
150   Marker$="=>"&CHR$(8)&CHR$(8)    ! Pointer arrow
160   Point=1                         ! Default selection
170   PRINTER IS 1                    ! Use CRT for menu display
180   GRAPHICS OFF
190   CONTROL 2,1;0                   ! PRT ALL off
200   CONTROL 1,4;0                   ! DISPLAY FCTNS off
210   OUTPUT KBD;Clear$;                ! Clear CRT
220   !
230   PRINT "Use shift and knob to move marker"
240   PRINT "   Selection 1"          ! Display menu
250   PRINT "   Selection 2"
260   PRINT "   Selection 3"
270   PRINT "   Selection 4"
280   PRINT "   Selection 5"
290   PRINT "   Selection 6"
300   PRINT "   Selection 7"
310   PRINT TABXY(1,Point);Marker$;   ! Display starting marker
320   !
330   ON KNOB .2 GOTO Move_pointer    ! Enable knob
340 Spin:   GOTO Spin                 ! Wait for knob interrupt
350   !
360 Move_pointer:   !
370   IF KNOBY>0 THEN                 ! Check knob direction
380     Point=Point+1
390   ELSE
400     Point=Point-1
410   END IF
420   IF Point<1 THEN Point=7         ! Keep pointer within limits
430   IF Point>7 THEN Point=1
440   OUTPUT KBD;Home$;                 ! Home the display
450   PRINT "  ";                     ! Erase old marker
460   PRINT TABXY(1,Point);Marker$;   ! Display new marker
470   GOTO Spin
480   !
490   END
```

The program starts by declaring and initializing the variables. The "clear" and "home" sequences should look familiar to you by now. The **Marker$** string is a contrived arrow followed by two backspace characters. The backspace characters return the print position to the beginning of the arrow each time it is displayed. This facilitates the erase operation that is part of moving the arrow.

After the display is cleared, the menu selections are printed. This is done only once, since the choices do not include any changing parameters. The TABXY function is used to position a marker to the left of the default selection. Then the knob is enabled, and the program sits in an idle loop waiting for an interrupt from the knob.

When the knob is turned, program execution branches to the pointer-moving routine. In this example, the amount of knob movement is not used, only its direction is extracted from the KNOBY function. It is possible to add an algorithm that accumulates the counts from the knob so that a fixed amount of rotation is needed to move the pointer. Such an improvement would give a more positive "linkage" between the knob and the display, but is not necessary to this demonstration.

The pointer value is stored in the variable `Point`. This variable is increased or decreased depending upon the direction of knob rotation. After the variable is updated, it is necessary to keep it within the limits of the available selections. The option used here was to "wrap around" when the pointer reached either end of the list. Another option is to "freeze" the pointer when it reaches an end position. To do this, lines 420 and 430 would be modified as follows:

```
420   IF Point<1 THEN Point=1
430   IF Point>7 THEN Point=7
```

After the pointer value is updated, the display must be changed to reflect the new value. First, the display is returned to home position. Although the knob no longer scrolls the display, the scrolling keys are still active. They may have been pressed (perhaps accidentally) and moved the display out of position. Since the print position is always at the beginning of the old pointer, that pointer can be erased by printing two blanks. The new pointer is then printed using a TABXY function. Notice that end-of-line sequences are not needed or desired. All the PRINT statements used in this updating process use a trailing semicolon to suppress the EOL sequence.

In this example, the x-coordinate was always 1. If needed, the x-coordinate is available in the TABXY function to work with multi-column displays.

Assumed, but not shown, is an ON KEY statement that would start the selected process. This key would branch to a routine that cleared the display, turned off the knob, and used the variable `Point` in a SELECT or ON statement to access the chosen routine.

8 Efficient Use of the Computer's Resources

Every model of computer has certain characteristics which can result in better performance, provided the programmer knows what those characteristics are and how he can take advantage of them. This chapter consists of a potpourri of such items.

Data Storage

It is usually desirable to minimize the usage of computer memory and mass storage. This section describes how much space is required to store various types of data, which will help you in using your storage resources for the best possible utilization.

Data Storage in Read/Write Memory

There are five data types on this computer: REAL, INTEGER, COMPLEX, strings, and I/O path names. The memory occupied by data is made up of two parts: the memory it actually takes to hold the intended information, and the memory that the system uses to keep track of the information's location and form (this is called overhead). Strings, INTEGERs, COMPLEXs and REALs can be declared either as simple variables or as arrays. Arrays take different amounts of overhead than simple variables, but each element of an array uses the same amount of memory that a corresponding simple variable uses to actually store information.

The overhead required for any given symbol is kept in three tables:

- the symbol table
- the token table
- and the dimension table

The symbol table contains pointers to the value area, where the actual information is kept, and to the other two tables. The token table contains the names of the various symbols. The dimension table contains length information for strings and arrays, and is not used for numeric scalars. The tables are not constructed in single units as symbols are added and deleted. Rather, as new space is required, the system will first look to see if there are any unused entries in the tables—if new space is allocated, usually enough for several entries is allocated. For instance, the symbol table is built in increments of five entries on Series 200/300/400 computers and increments of eight entries on Series 700 computers.

Symbol Table Overhead:	10 bytes per symbol for Series 200/300/400 computers. 20 bytes per symbol for Series 700 computers.
Token Table Overhead:	Number of characters in the name + 1 (if the above number is odd, it is rounded up to an even number). Note that the name for I/O path names, strings, and functions includes the @, $, and FN, respectively.
Dimension Table Overhead:	For arrays:
	3 bytes (total size) on Series 200/300/400 and 8 bytes on Series 700
	1 byte (number of dimensions) on Series 200/300/400; 4 bytes (number of dimensions) Series 700.
	4 bytes for each dimension (for the lower bound, and the size of each dimension) on Series 200/300/400. 8 bytes for each dimension on Series 700.
	For strings:
	2 bytes (maximum length) on Series 200/300/400; 4 bytes (maximum length) on Series 700

For string arrays:

All of the normal array overhead, plus two bytes (Series 200/300/400) or four bytes (Series 700) for the maximum allowed length of an element

Line labels, COM labels, and subprograms are considered to be symbols, and occupy space in both the symbol and token tables. Line numbers used *in* statements, like GOTO 20, also occupy space in the symbol table.

Every subprogram (or context) has its own set of tables. In addition, there is a global set of COM tables where all information concerning COM blocks is kept. Symbols that belong to a COM block will occur in both the COM tables and in any local tables in which that COM block is declared. Since each context may define the names by which it refers to COM block variables, there will be no entry in the COM token table for each variable, but an entry in the COM token table *will* occur for COM labels.

ALLOCATEd variables require four bytes of overhead in addition to the overhead already discussed for the symbol, token, and dimension tables.

Note In HP BASIC/UX 700, all variables must be 8-byte aligned. Therefore, any variable that is not a multiple of 8 bytes (in the information storage part) gets padded until it is a multiple of 8 bytes.

The following table summarizes the storage requirements for various data types. This table does not show the extra requirements just mentioned for ALLOCATEd and COM variables.

Data Type Storage Requirements

Type	Overhead	Information Storage
Simple INTEGER	(Series 200/300/400) 10 bytes + name overhead	(Series 200/300/400) 2 bytes
	(Series 700) 20 bytes + name overhead	(Series 700) 2 bytes storage and 6 bytes padding
Simple REAL	(Series 200/300/400) 10 bytes + name overhead	(Series 200/300/400) 8 bytes
	(Series 700) 20 bytes + name overhead	(Series 700) 8 bytes
Simple COMPLEX	(Series 200/300/400) 10 bytes + name overhead	(Series 200/300/400) 16 bytes
	(Series 700) 20 bytes + name overhead	(Series 700) 16 bytes
Simple string	(Series 200/300/400) 12 bytes + name overhead	(Series 200/300/400) 1 byte per char. up to declared length, padded to even number of chars. + 2 bytes of length information
	(Series 700) 24 bytes + name overhead	(Series 700) (1 byte per char. up to declared length, padded to a multiple of 4 bytes + 4 bytes of length information)—with the total padded to a multiple of 8 bytes
I/O path name	(Series 200/300/400) 10 bytes + name overhead	(Series 200/300/400) 190 bytes
	(Series 700) 20 bytes + name overhead	(Series 700) 196 bytes

Data Type Storage Requirements (continued)

Type	Overhead	Information Storage
INTEGER array	(Series 200/300/400) 14 bytes + name overhead + 4 bytes per dimension	(Series 200/300/400) 2 bytes per element
	(Series 700) 32 bytes + name overhead + 8 bytes per dimension	(Series 700) 2 bytes per element; total padded to a multiple of 8 bytes
REAL array	(Series 200/300/400) 14 bytes + name overhead + 4 bytes per dimension	(Series 200/300/400) 8 bytes per element
	(Series 700) 32 bytes + name overhead + 8 bytes per dimension	(Series 700) 8 bytes per element
COMPLEX array	(Series 200/300/400) 14 bytes + name overhead + 4 bytes per dimension	(Series 200/300/400) 16 bytes per element
	(Series 700) 32 bytes + name overhead + 8 bytes per dimension	(Series 700) 16 bytes per element
String array	(Series 200/300/400) 16 bytes + name overhead + 4 bytes per dimension	(Series 200/300/400) (1 byte per char. up to declared length, padded to even number of chars. + 2 bytes of length information)—per array element
	(Series 700) 36 bytes + name overhead + 8 bytes per dimension	(Series 700) (1 byte per char. up to the declared length, padded to a multiple of 4 bytes + 4 bytes of length information)—per array element, then the total is padded to a multiple of 8 bytes

Data Storage on Mass Memory Devices

The storage size of data on mass storage media is similar to the amount of memory that data takes internally, except that no overhead is required (on BDAT files). On HP BASIC/UX 700, data is aligned on disk (or other mass storage media) in exactly the same way as it is on HP BASIC/UX 300/400. That is, the memory alignment specifics, detailed in the previous table, don't apply to data storage on mass storage media. Arrays and single values are interchangeable on mass storage—no distinguishing information is kept on the media.

INTEGERs (and INTEGER arrays)	2 bytes (per element)
REALs (and REAL arrays)	8 bytes (per element)
COMPLEXs (and COMPLEX arrays)	16 bytes (per element)
Strings (and string arrays)	4 bytes + 1 byte per char up to current length, padded to even number of chars. (per element)

For ASCII files, all information is converted to string (or ASCII) form, and a two-byte length field is tacked onto the front of every field.

INTEGERs (and INTEGER arrays)	2 bytes + 1 byte per digit (per element)
REALs (and REAL arrays)	2 bytes + 1 byte per digit (per element)
COMPLEXs (and COMPLEX arrays)	2 bytes + 1 byte per digit (per element)
Strings (and string arrays)	2 bytes + 1 byte per char (per element)

Comments and Multi-character Identifiers

Self-documenting features such as in-line comments and multi-character variables and line labels are useful because of the benefits to be reaped in terms of developing, testing, debugging, and maintaining programs. They do take extra memory, but this shouldn't be a problem if you keep the following points in mind.

Comments take 1 byte of memory for every character in the comment. If memory space becomes a problem, many people resort to keeping two copies of their programs around—one fully commented to use as reference material, and the other uncommented to use as the "production version," which is the one that is actually used.

Multi-character identifiers are only spelled out in their entirety once—not every time they are used. The program actually stores pointers whenever a reference to the identifier is used, so using short identifiers won't result in any appreciable savings in memory used.

Variable and Array Initialization

Care should be taken to initialize any variables before using them in an expression (on the right hand side of an =, as a left-hand subscript in a function or subprogram parameter list, as an argument to a built-in function, or in a PRINT/OUTPUT/DISP list). The system will set variables to zero, strings to null, and I/O path names to undefined at program prerun, but depending upon default settings is considered bad programming practice and could lead to subtle errors. For instance, the first time a certain line is executed, the variables used may be assumed to be zero because of the prerun operations. Once this assumption has been made, the danger is that the programmer will branch back to the same section of code and forget that the zeroing process has not been performed—an error may result that didn't occur previously.

Mass Memory Performance

Program Files

There are two ways to store programs—they can be saved either as ASCII source strings using the SAVE command, or they can be stored in an intermediate form that the BASIC language system understands using the STORE command.

Note Program files created by the STORE command come in two different forms: PROG files (created from HP BASIC/UX 300/400 and HP BASIC/WS) and PROG2 files (created from HP BASIC/UX 700). PROG and PROG2 files are *not* interchangeable: HP BASIC/UX 700 will LOAD PROG files—transparently, but more slowly than an equivalent PROG2 file—however, HP BASIC/UX 300/400 and HP BASIC/WS will *not* recognize or LOAD PROG2 files. Refer to the *HP BASIC Porting and Globalization* manual for more details.

If the time it takes to load the program is important, always use the STORE command to store the program instead of the SAVE command. The LOAD command, which reads in files created by the STORE command, will execute about fifty times faster than the GET command. This is because the LOAD command does not require that the information on the file be processed in any way. Since the program is already in the form the system needs it in, all that is necessary is to funnel the program directly into memory as fast as the disk can spin (assuming an interleave of one, and that HP BASIC/UX 700 is loading PROG2, not PROG, files).

SAVE files, on the other hand, require that the system parse and check the lines as they are read, just the same as if a user had typed them in from the keyboard. Consequently, the speed at which the program gets loaded into memory with the GET command will be drastically slower than the LOAD command. Using the Model 226 and 236 internal drives as an example of the relative speeds, a typical 8-Kbyte program will take about 30 seconds to GET, but only about one second to LOAD.

One advantage of the GET/SAVE commands is that it is possible to deal with programs as string data.

Data Files

As with program files, there are different types of data files: ASCII, BDAT, and HP-UX. ASCII files require that all data be in string form, while BDAT and HP-UX files are interpreted as internal data representations, if the ASSIGN statement requests FORMAT OFF.

When reading or writing data to an ASCII file, the number formatter is required to convert the data in between its internal representation and its ASCII form. When reading or writing data to a "FORMAT OFF" BDAT or HP-UX file, the data may stream directly back and forth with no conversion required. Using the Model 226 and 236 internal drives as an example, an 8K-element REAL array (64K bytes) may take around 200 seconds to write in an ASCII file, while the same array will only take about 5 seconds to write to a BDAT file.

The primary benefit of the ASCII data file is the transportation of data between different models of Hewlett-Packard computers and terminals and between disks used with different language systems. HP-UX files are also interchangeable with other computers that support the HFS file system.

Benchmarking Techniques

This section discusses the techniques used to determine how fast various operations execute. Ideally, you should separate the measurement time from the elapsed time:

```
10   T1=TIMEDATE
20   T2=TIMEDATE
30   PRINT T1-T2;"seconds used to read clock"
40   END
```

In actuality, the clock has a resolution of only 20 ms in HP BASIC/WS and HP BASIC/UX 300/400, or 10 ms in HP BASIC/UX 700, therefore, you usually won't be able to time this operation.

Next, most operations are performed inside a loop in order to be able to time operations that are faster than the resolution of the clock. This also tends to "smooth out" varying system overhead characteristics. The following program measures the time consumed by incrementing the INTEGER variable I in the FOR loop.

```
10   INTEGER I
20   T1=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   T2=TIMEDATE
60   PRINT T2-T1;"seconds of loop overhead"
70   END
```

A certain amount of time used in computational operations will involve moving information around. The time will be different depending upon the data type of the information—string, REAL, COMPLEX or INTEGER—being manipulated. Also, differences in string length affect the computational time. The following program measures the time consumed by incrementing the INTEGER variable *I* and by assigning the value of the REAL variable *B* to the REAL variable *A* in the FOR loop.

```
10   REAL A,B,C
20   INTEGER I
30   B=PI
40   T1=TIMEDATE
50   FOR I=1 TO 10000
60   A=B
70   NEXT I
80   T2=TIMEDATE
90   PRINT T2-T1;"seconds of loop overhead"
100  END
```

Finally, you can measure the actual time required for arithmetic operations. The following program measures the time required for the addition operation. Specifically, the addition operation in line 120 is the one being measured. The time required to perform many of the arithmetic operations may vary depending upon the data types and values (number of digits and relative magnitudes) of the two operands.

```
10   REAL A,B,C
20   INTEGER I
30   B=PI*1.E+53
40   C=EXP(SQR(2)^13.81)
50   PRINT "B=";B,"C=";C
60   T1=TIMEDATE
70   FOR I=1 TO 10000
80     A=B
90   NEXT I
100  T2=TIMEDATE
110  FOR I=1 TO 10000
120    A=B+C
130  NEXT I
140  T3=TIMEDATE
150  Op_time=DROUND(T3-T2-T2+T1,3)
160  PRINT Op_time*100;"us. per operation"
170  END
```

You can modify line 120 of the program above to measure other arithmetic operations, for example:

- subtraction
- multiplication
- division
- exponentiation

INTEGER Variables

We have seen in the first section of this chapter that INTEGER variables don't take as much memory as REAL variables (2 bytes instead of 8 on HP BASIC/WS and HP BASIC/UX 300/400). Now we will discover that some operations with INTEGERs are much faster than the same operations with REALs.

Note The following results are not always significant to programming in HP BASIC/UX 700, however, if you intend to run your programs in HP BASIC/UX 300/400 (on Series 300/400 computers), as well as in HP BASIC/UX 700, these techniques may prove useful.

Minimum and Maximum Values

The INTEGER variable type may store any whole number from −32 768 to +32 767 inclusive.

Mathematical Operations

There are two sets of math routines provided for the MOD, DIV, +, −, and * operations: REAL and INTEGER. Depending upon the types of the operands used, the execution times for these operations may vary widely. The tradeoffs are:

INTEGER math is the faster of the two, since it doesn't require as much "work." This is because:

1. There are only two bytes of data to process instead of eight.
2. Operations do not have to deal with a combination of mantissa and exponent.
3. The results don't have to be normalized.
4. INTEGER math can be done directly in the hardware.

REAL math, though slower, is generally more widely used because it allows numbers with fractional parts to be analyzed. REAL numbers carry about 16 decimal digits of precision and have an exponent range of −308 to +308.

For instance, suppose you want to compute your monthly pay. Assume that you're making $5.17 an hour, that you work twenty-four days per month and that you work 14 hours per day. The calculation that you would use is `5.17*24*14` or $1737.12. In this problem, you definitely want your computer to use REAL precision math (or you'll lose 17 cents per hour!) even though you're only using 6 of the 16 digits available.

The computer will pick whatever math routines it needs to solve the current problem. However, the programmer can exercise control over which math routines get executed if the following rules are understood.

- INTEGER math is used if both arguments of a MOD, DIV, *, +, or − operation are of type INTEGER. If the results of the operation cannot be stored in an INTEGER, then an error is generated (INTEGER overflow).
- REAL math is used if either or both arguments of a MOD, DIV, *, +, or − operation is of type REAL. If one of the arguments is of type INTEGER, then that argument is first converted to REAL.
- REAL math is always used for exponentiation (^) and division (/).

The following table gives some approximate time comparisons between INTEGER and REAL operations for +, −, and *. The times are approximations because REAL math routines do different things depending upon the values of the operands. All times shown here were found on operations with numbers having no fractional parts. The multiplication times were found for operands in the range of −200 to +200.

Note The following execution times are for a Series 200 computer with an MC68000 processor running at 8 MHz. They will be significantly decreased on computers with higher clock rates or floating-point math hardware (HP 98635 math card or MC68881/2 coprocessor or Series 700 computer).

Approximate Execution Times: INTEGER vs. REAL

Operation	REAL	INTEGER
MOD	160 μs	91 μs
DIV	352 μs	88 μs
Addition	142 μs	68 μs
Subtraction	174 μs	68 μs
Multiplication	152 μs	77 μs

Multiplication, like most math operations, will execute faster on INTEGER values. However, bear in mind that it's much easier to get an INTEGER overflow on multiplications than on additions and subtractions. For instance, 200*200 will give an INTEGER overflow. If you are performing multiplication on INTEGERs, you should carefully examine your program to ensure that the range of your answers doesn't force you to use REALs, even if the requirement for fractional precision doesn't.

Loops

In general, any FOR/NEXT loop using an index which has been declared to be an INTEGER will execute about 2.4 times faster than a loop whose loop counter is a REAL. Type in the two programs below and run them to see the difference.

```
10   REAL I
20   T0=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   PRINT TIMEDATE-T0;"seconds"
60   END
```

Time is about 1.67 seconds.

```
10   INTEGER I
20   T0=TIMEDATE
30   FOR I=1 TO 10000
40   NEXT I
50   PRINT TIMEDATE-T0;"seconds"
60   END
```

Time is about .69 seconds.

Bear in mind that the 2.4 speed improvement is only on the time devoted to actually incrementing and testing the loop variable (in these examples, I). So, any loop that iterates for 10 000 times will run about a second faster if the index is an INTEGER instead of a REAL. Now, saving a second on a loop that executes 10 000 times may not sound like much by itself, and it's not. But what if that loop is nested inside *another* one that executes 10 000 times? Now your time savings is 10 000 seconds, or two hours and forty-five minutes! Just for declaring the loop counters to be INTEGER.

Naturally, making a loop index an INTEGER will only work if the loop is not stepping in fractions, and if the minimum and maximum values of the loop index do not exceed the range of −32 768 to +32 767.

Array Indexing

Accessing individual array elements is faster if the variables or expressions giving the indices into the array are INTEGER instead of REAL. This is because the system has to convert floating-point numbers into an INTEGER in order to find the offset from the beginning of the array. If the index is already in INTEGER form, the conversion isn't necessary. The following example illustrates this point.

```
10   REAL I
20   DIM A(1:1000)
30   X=17.568
40   T0=TIMEDATE
50   FOR I=1 TO 1000
60      A(I)=X
70   NEXT I
80   PRINT TIMEDATE-T0;"seconds"
90   END
```

```
10   INTEGER I
20   DIM A(1:1000)
30   X=17.568
40   T0=TIMEDATE
50   FOR I=1 TO 1000
60      A(I)=X
70   NEXT I
80   PRINT TIMEDATE-T0;"seconds"
90   END
```

You will find a difference of .14 seconds between the two programs' execution times, due to a combination of the loop counter being INTEGER and the INTEGER indexing of the array. Again, if you're operating on a much larger array, or if you're working on a multi-dimensional array, this number can become noticeable.

REAL and COMPLEX Numbers

This section describes details of using and storing REAL numbers. The information can generally be applied to COMPLEX numbers, since each COMPLEX number is composed of two REAL numbers.

Minimum and Maximum Values

The minimum REAL number that can be entered in HP BASIC is approximately $\pm 2.225\ 073\ 858\ 507\ 202 \times 10^{-308}$ (The MINREAL function returns this value.)

The maximum REAL number that can be entered in HP BASIC is approximately $\pm 1.797\ 693\ 134\ 862\ 315 \times 10^{308}$ (The MAXREAL function returns this value.)

A REAL number can also have the value zero.

On Series 700 computers, denormalized numbers may occur as a result of numeric operations if *underflow_mode* is set to IGNORE. (Refer to the "Controlling Underflow Handling (*underflow_mode*) (Series 700 Only)" section in the "Creating Environment and Autostart Files" chapter of *Using HP BASIC/UX* for more information on *underflow_mode*).
The minimum REAL number that will result from performing numeric computations in HP BASIC/UX 700 with *underflow_mode* set to IGNORE is $4.940\ 656\ 458\ 412\ 465 \times 10^{-324}$

With `underflow_mode=ignore` it is recommended (for compatibility with HP BASIC/UX 300, HP BASIC/WS, and the other `underflow_mode` settings) that you check the absolute value of potentially denormalized numbers against `MINREAL` and set them to 0 if they are denormalized. For example,

```
IF ABS(Small_num)<MINREAL THEN Small_num=0
```

The IEEE 754 Math specification defines the values `INF` (infinity) `NaN` (not a number). These numbers cannot normally be created by BASIC, however may appear when data is read from interfaces, files, pipe, or other applications that can create these numbers. With HP BASIC/UX 700 computations using `INF` and `NaN` give these results:

±`INF`	will produce ±`INF`
signaling `NaN`	will produce a signaling `NaN`
non-signaling `NaN`	will produce `ERROR 21  Exception in math operation`

With HP BASIC/UX 700 printing `INF` and `NaN` give these results:

+`INF`	will produce the text " ++"
−`INF`	will produce the text "−−−"
signaling `NaN`	will produce the text "?"
non-signaling `NaN`	will produce `ERROR 21  Exception in math operation`

Earlier versions of HP BASIC will give errors when you try to print or otherwise format `INF` or `NaN` as ASCII, but will usually not generate errors when computations are done with them.

Type Conversions

Earlier, it was mentioned that any time a MOD, DIV, *, +, or − operation is performed on two numbers of different type (one INTEGER, and one REAL), a type conversion has to take place to convert the INTEGER to a REAL. This section will address other situations where type conversions have to take place.

Any time an INTEGER is used in an exponentiation or division operation, it must first be converted to a REAL.

All of the following functions require a REAL argument (with the exception of VAL and RND), and all of them return a REAL value (with the exception of RANDOMIZE). If an INTEGER is passed in, or if the result is to be stored in an INTEGER, then the appropriate type conversion must be made: EXP, LGT, LOG, RANDOMIZE, SQRT, DROUND, RND, ACS, COS, ASN, SIN, ATN, TAN, VAL. Note that many of the previously mentioned functions also take COMPLEX arguments and return COMPLEX arguments. These functions are: EXP, LGT, LOG, SQRT, ACS, COS, ASN, SIN, ATN, TAN.

All of the comparison operators (=, <>, <, >, <=, >=) will return INTEGER values (0 or 1) but will accept either INTEGERs or REALs as arguments. The only comparison operators allowed with COMPLEX values are = and <>. For more information on comparisons of COMPLEX values refer to the "Handling Errors" chapter of the *HP BASIC Programming Guide*. The logical operators AND, EXOR, OR, and NOT will convert any arguments to the INTEGER values 0 or 1 before the operation is performed, and an INTEGER 0 or 1 will be returned.

The binary bit functions (BINAND, SHIFT, ROTATE, BINIOR, BINCMP, BIT, BINEOR) require INTEGER inputs and provide INTEGER outputs. Type conversions will be performed if REALs are supplied as inputs, or if the results are to be stored in a REAL variable.

SGN returns an INTEGER (−1, 0, 1) regardless of the type of the argument passed to it. ABS and INT return the type of the argument that's passed to them.

If two INTEGERs are used to perform a MOD, DIV, *, +, or − operation, but the result is to be stored in a REAL variable instead of an INTEGER, then the result must be converted from INTEGER to REAL.

Constants

All constants that are within the range of −32 767 to 32 767 that aren't entered with a decimal point or an "E" (for scientific notation) are stored in the computer as INTEGERs. Integer constants should always be used with INTEGER variables, but if they are used with REAL variables they will have to be converted to REAL first. This operation will slow down the execution of the program, as shown in the previous section. Any numbers entered with decimal points (1.0, 3., .7, etc.), with an E (1E−304, .2E48, 0E0, etc.), or outside the valid INTEGER range (40000, −75986, etc.) will be stored as REAL constants.

Polynomial Evaluations

The polynomial can waste much computer time because programmers tend to pick the most obvious, and also the most time-consuming, method of evaluating them. Polynomials are usually written mathematically as:

$$y = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

or

$$y = \sum_{i=0}^{n} a_i x^i$$

Hence the temptation is strong to evaluate a polynomial on a computer as:

```
2000 DEF FNPoly(X,Coefficient(*),INTEGER N)
2010 INTEGER I
2020 Y=0
2030 FOR I=0 TO N
2040    Y=Y+Coefficient(I)*(X^I)
2050 NEXT I
2060 RETURN Y
2070 FNEND
```

In the above program, there are N+1 additions, N+1 multiplications, N+1 exponentiations, and N+1 INTEGER to REAL conversions (I is converted to a REAL when the exponentiation operation is performed). Now suppose the polynomial is written in the equivalent form:

$$y = a_0 + x(a_1 + x(a_2 + \cdots + X(a_n)\cdots))$$

Then the corresponding program would be:

```
2000 DEF FNPoly(X,Coefficient(*),INTEGER N)
2010 INTEGER I
2020 Y=Coefficient(N)
2030 FOR I=N-1 TO 0 STEP -1
2040    Y=Coefficient(I)+X*Y
2050 NEXT I
2060 RETURN Y
2070 FNEND
```

Now there are only N additions and N multiplies, and NO exponentiations or INTEGER to REAL conversions! For example, if the polynomial is of order 10, it is 70 milliseconds faster to use the nested multiply method to evaluate the polynomial than to use exponentiation. If you're plotting a thousand points on a graph, nested multiplication will save you more than a minute!

Logical Comparisons for Equality on REAL Numbers

Don't do it.

If you are performing mathematical operations on REAL numbers, and then comparing them for equality, the chances are that they will never come up equal. For example, in the previous section on polynomial evaluation, you can pass the same value for X and the same coefficient array to each of the two functions, and although the results will look equal when you print them out, they won't show equality if you compare them. (Try it and see.) A shorter example is to execute this expression from the keyboard:

```
.1+.1+.1+.1+.1+.1+.1=.7
```

The 0 at the bottom of the screen means that the computer doesn't consider the two numbers to be equal. This is characteristic of the way binary math works.

Any REAL numbers should be rounded first before being tested for equality. This is one of the purposes of the DROUND function.

```
DROUND(.1+.1+.1+.1+.1+.1+.1,12)=DROUND(.7,12)
```

After rounding both numbers to 12 digits, the computer will now accept them as being equal. See the "Numeric Computation" chapter in the *HP BASIC Programming Guide* for more discussion on the comparison of REAL numbers.

Saving Time

Multiply vs. Add

It is always faster to add a number to itself than it is to multiply it by 2. For instance, to perform `2*PI` a thousand times takes .22 seconds, while to perform `PI+PI` a thousand times takes .13 seconds.

However, if you want to multiply by 3, that is faster than adding the number three times. `3*PI` executed a thousand times takes about the same as `2*PI` (.22 seconds), but adding `PI+PI+PI` a thousand times takes about .28 seconds.

Exponentiation vs. Multiply and SQRT

Exponentiation is very slow when compared to other mathematical operations. The results are worth the wait when the exponent has a fractional part; raising a REAL number to a REAL power is a complex operation. However, time can be saved if you are alert to some special cases. The most common examples are squaring a number or finding a square root. Using `SQRT(X)` takes only about one-fourth the time required by the expression `X^.5`. Even more dramatic savings can be gained when raising numbers to an integer power. Using `X*X` yields a 22-to-1 time savings over the expression `X^2`. When using powers greater than 2 or 3, the expressions needed to achieve the repeated multiplication can be somewhat cumbersome, and may not even fit on a program line. However, repeated multiplication is so much faster than exponentiation that time savings can be realized (for powers up to 14) even if a FOR ... NEXT loop has to be added to repeat the multiplication.

Array Fetches vs. Simple Variables

It takes more time to access an array element than it does a simple variable. This is because the address of the array element has to be computed from the starting address of the array and the offset within the array based on the specified indices. A simple variable's address does not require this computation.

Thus, if you access a given array element often enough, especially within a loop, it will often be faster to store the array element into a simple variable and then operate on the simple variable.

Time to fetch a simple variable and store it:	136 μs
Time to fetch an array variable and store it:	260 μs
Difference:	124 μs

This means that it is faster to fetch three simple variables than it is to fetch two array elements.

Concatenation vs. Substring Placement

The concatenation operator (`&`) allows you to combine two or more strings to construct another string. This operation is discussed in the "String Manipulation" chapter of the *HP BASIC Programming Guide.* However, there is a special case that can be accomplished faster without the concatenation operator. This is the case where the new string is built by appending to the end of an existing string. For example, `A$=A$&B$`.

Concatenation works by constructing a temporary workspace in which all the components are assembled. Then the result is moved to its destination. In the example above, A$ is moved to a temporary workspace, B$ is appended to it, and the result is moved back to A$. Thus, the original contents of A$, which weren't changed, have been moved twice unnecessarily. A faster way to accomplish the same thing is:

```
A$[LEN(A$)+1]=B$
```

Another benefit of this approach is that the temporary workspace is not created. If memory is tight and A$ is very large, concatenation could create a memory overflow.

Using Floating-Point Math Hardware

This section describes the floating-point math hardware that is supported by BASIC. It shows how to enable and disable this hardware, also shows an example of how it improves the performance of numeric computations.

The HP 98635 Floating-Point Math Card (Series 200 and Model 310 Only)

This card is an optional printed-circuit board which plugs into the backplane of your computer. It contains a special integrated-circuit chip which performs floating-point math computations, rather than having the BASIC system software perform them. It provides significant speed improvements over the software method.

The MC68881/68882 Floating-Point Math Coprocessor

Series 300/400 computers may optionally be equipped with an MC68881 or MC68882 floating-point math coprocessor. These chips reside on the same board that contain the main CPU (such as a 68020). Not only do these chips provide an increase in *speed* of math calculations, but they also increase the *accuracy* of these calculations. These chips use 80-bit precision, rather than the 64-bit precision of the BASIC software math library (and the HP 98635, which also uses 64-bit precision). For instance, in a series of standard math tests, the RMS (root mean square) error in the 10 worst cases for the MC68881/68882 chips ranged from 0 to 0.37 bits of error. On the other hand, the BASIC software math library and HP 98635 card had 10 worst cases in the range of 0.33 to 4.2 bits of error.

While the BASIC math library and the HP 98635 Math Card produce identical results, these values may not agree with those obtained from using the MC68881/68882 coprocessor. This may only be noticeable when strict equality with the math library or 98635 card is required (which is not recommended). For strict compliance, disable the MC68881/68882 chip.

For BASIC/WS, the MCMATH binary provides MC68882-compatible floating-point math routines for the MC68040 processor in the HP 9000 Model 380 and 382 computers. (For BASIC/UX, these routines are installed as part of the core system.)

Floating-Point Hardware Supported

BASIC/WS supports either the MC68881 or MC68882 Floating-Point Math Coprocessor as well as the HP 98635 Floating-Point Math Card.

BASIC/UX supports the same floating-point math hardware as BASIC/WS. However, unlike HP-UX, BASIC/UX does not support the HP 98248 Floating-Point Accelerator Card. The hardware supported for each model of computer is provided in the following table.

Note With the release of HP-UX 7.0, BASIC/UX 300/400 is *not* supported on the HP 9000 Model 310 computer. However, BASIC/WS will continue to be supported on the Model 310 computer.

Floating-Point Hardware Supported by BASIC

Computer Model	Floating-Point Hardware
Series 200/310 (BASIC/WS *only*)	HP 98635A Floating-Point Math Card
320	HP 98635A or MC68881 Coprocessor
330, 332, 340, 345, 350, 360, 362, 370, 375, and 400	MC68881/68882 Coprocessor (the HP 98635A card is *not* supported on these computers)
HP 82324A High-Performance Measurement Coprocessor	HP 82327A Floating Point Unit (an MC68882)
380, 382, 425, 433	No floating-point hardware required. For BASIC/WS, the MCMATH binary provides MC68882-compatible floating-point software routines for the MC68040 processor. These routines are also included as part of the BASIC/UX system.
Series 700	All Series 700 computers have an internal floating-point processor.

BASIC will determine which floating-point math hardware is present and select the fastest one (unless it has been explicitly turned off).

Example

If you have the appropriate floating-point math hardware present, this program provides you with an example of the speed increase you can gain from it if you have a calculation intensive program. Note that the floating-point math capability is turned on and off using pseudo-select code 32 (see lines 140, 340, and 360).

```
100   REAL X,Y,Result
110   X=88.92
120   Y=25.135
130   Count=0                      ! Initialize the count variable.
140   CONTROL 32,2;1               ! Turn floating-point hardware on.
150   REPEAT
160     STATUS 32,2;Status         ! Status of floating-point hardware.
170     IF Status=1 THEN
180       PRINT "Seconds elapsed for the first calculation ";
190       PRINT "with the floating-point hardware on."
200       PRINT
210     ELSE
220       PRINT "Seconds elapsed for the second calculation ";
230       PRINT "with the floating-point hardware off."
240       PRINT
250     END IF
260     Start$=TIME$(TIMEDATE)     ! Save the starting time.
270     FOR I=1 TO 100000
280       Result=ACS(SIN(I*PI/50000))
290     NEXT I
300     Finish$=TIME$(TIMEDATE)    ! Save the stopping time.
310     PRINT USING "DDDD.DD";TIME(Finish$)-TIME(Start$) ! Time elapsed.
320     PRINT
330     Count=Count+1              ! Increment the count variable.
340     CONTROL 32,2;0             ! Turn the floating-point hardware off.
350   UNTIL Count>1
360   CONTROL 32,2;1       ! Turn the floating-point hardware back on.
370   END
```

Your display will contain the following results if you run the above program on a Model 350.

```
Seconds elapsed for the first calculation with the floating-point hardware on.

 22.60

Seconds elapsed for the second calculation with the floating-point hardware
off.

 187.24
```

Internal Cache Memory

Most of the HP 9000 Series 300 computers use MC68020, MC68030, or MC68040 processors, all of which have on-chip, high-speed cache memory. This memory serves as storage for machine-instruction sequences, typically allowing the processor to decrease the amount of off-chip memory accesses, which increases program-execution speed.

Note The HP 82324 High-Performance Measurement Coprocessor, which uses an MC68030 processor, also has internal cache memory. The HP 82300 Measurement Coprocessor, the Model 310, and the Series 200 computers do not have internal cache memory.

Enabling and Disabling Cache Memory (BASIC/WS and BASIC/DOS only)

BASIC/WS supports enabling and disabling cache memory. BASIC/DOS also supports this for the HP 82324 High-Performance Measurement Coprocessor only. BASIC/UX does not support this.

You can determine whether or not cache memory is currently enabled with this statement:

```
STATUS 32,3;Cache_on
```

If the variable called `Cache_on` is assigned a value of 1, then cache is currently enabled (this is the default condition). If it is assigned a value of 0, then cache is disabled.

If the cache feature is enabled, but you want to disable it, you can do so with this statement:

```
CONTROL 32,3;0
```

If you want to re-enable this feature, execute this statement:

```
CONTROL 32,3;1
```

Selecting the Cache Mode (BASIC/WS and MC68040 only)

The MC68040 processor (used in Models 380 and 382) implements internal cache memory slightly differently than the MC68020 and MC68030. The procedure for turning cache on and off is the same as described in the previous section, and the MC68040 defaults to “cache on”, as do the MC68020 and MC68030. However, the MC68040 implements two cache memory modes:

- The **copy-back** cache mode is the default for the MC68040 processor. In copy-back mode, “dirty” cache entries are not written to physical memory until the cache line is needed for other data, or the cache is explicitly flushed. (A “dirty” cache entry is one that differs from the corresponding data in physical memory.)
- The **write-through** cache mode is functionally the same as cache operation for the MC68020 and MC68030 processors. In this mode, when the processor writes to the internal cache, physical memory is written to at the same time.

The copy-back mode provides additional speed for most operations over the write-through mode. However, you may want to select the write-through mode for compatibility with the MC68020 and MC68030 processors. To do this, HP BASIC/WS provides an additional register: CONTROL/STATUS 32,6.

To select write-through caching, execute:

```
CONTROL 32,6;0
```

To select copy-back caching (MC68040 only), execute:

```
CONTROL 32,6;1
```

To determine the current mode (0 = write-through, 1 = copy-back):

```
STATUS 32,6;Cache_mode
```

Saving Memory

The ALLOCATE and DEALLOCATE statements can be used to make efficient use of memory space in certain applications. They are useful in programs that contain a number of large variables that are not all needed simultaneously. For example: during data collection, a large string array is needed; during data processing a large numeric look-up table is needed; and during output formatting, a string array is needed again. Because a mass storage device is used to hold the data between processes, the same memory area can be used for all these arrays.

To use ALLOCATE effectively, it is necessary to understand how the system reclaims areas that have been DEALLOCATED. Space for allocated variables is built using a stack discipline. The DEALLOCATE statement marks a space as unused. Unused space can be reclaimed only if it is the *last* space on the stack. There are two operations that use space in this stack. One is ALLOCATE, and the other is ON <event>.

Keeping other allocated variables from blocking deallocated space is relatively simple. If you have only one allocated variable at any given time, this is not a problem. If you have allocated variables in existence simultaneously, ALLOCATE them in the opposite order of the DEALLOCATE statements. Think of this in the same way you would think about nesting FOR ... NEXT loops.

Preventing blockage by ON conditions is more complicated. ON conditions, with one exception, create control blocks that are *permanent* entries on the stack. As soon as you declare an ON (or OFF) condition, all the previous entries on the stack are "locked in" for the duration of the context and cannot be reclaimed. Therefore, all the control blocks should be created *before* any variables are allocated. Once a control block is created, it will be used by all subsequent ON and OFF statements that refer to the same resource. A good technique is to include an OFF statement for each desired event before allocating any variables.

The exception mentioned above is an ON condition declared for an I/O path name. This includes ON END, ON EOT, and ON EOR. For these, subsequent ON and OFF statements behave as previously described. However, if the I/O path is closed, any control blocks associated with the path are marked as unused. This has two implications. One, the reclaiming of the stack will not be blocked after the I/O path is closed. Two, you cannot force the system to leave these control blocks at the beginning of the stack. Here is an example:

```
200  ASSIGN @File to "FRED"
210  ON END @File GOTO Label1
220  ALLOCATE Array(255,4)
 .
 .
 .
600  ASSIGN @File TO "SUSAN"
610  ON END @File GOTO Label2
620  DEALLOCATE Array(*)
```

At first, the array and control block are allocated in the proper order. The ASSIGN statement in line 600 closes the original path and opens a new path with the same name. When the ON END control block for the new path is created, it it placed after the array on the stack. Therefore, no memory space can be recovered by deallocating the array.

Releasing Memory Volumes

This section describes some subtleties that occur when creating memory volumes and subsequently reclaiming the memory they use.

Note that this discussion does *not* pertain to the BASIC/UX system, since it handles memory volumes differently.

Background (Creating Memory Volumes)

You can create a memory volume (a “disk” in memory) with the INITIALIZE statement; for instance, this statement creates a memory volume (at unit 0) with a size of 12 sectors (256 bytes each):

```
INITIALIZE ":MEMORY,0,0", 12
```

The size of the memory volume is $256 \times n+14$, where n is the number of sectors requested in the INITIALIZE statement (unless n is zero).

Reclaiming Memory (by Deleting a Memory Volume)

Memory used for this volume can be reclaimed without having to use SCRATCH A, which results in the loss of typing-aid softkey definitions and other customizations. However, you must keep in mind that memory can *only* be reclaimed if no binaries have been loaded after initializing the memory volume. For instance, to recover the memory volume created above, you would execute this command:

```
INITIALIZE ":MEMORY,0,0", 0
```

This, in effect, is equivalent to re-initializing the volume to 0 sectors to remove it from memory.

A Subtler Example (BASIC/WS only)

Memory volumes are allocated in a “mark and release” stack (for those of you familiar with Pascal). What this means is that you can only reclaim the memory if other subsequently created memory volumes have been reclaimed. The program on the next page illustrates how this works.

```
100  PRINT SYSTEM$("AVAILABLE MEMORY")  ! Show amount of memory available.
110  INITIALIZE ":,0,0",4               ! Create unit 0,
120  INITIALIZE ":,0,1",4               !        unit 1,
130  INITIALIZE ":,0,2",4               !        unit 2 (4 sectors each).
140  PRINT SYSTEM$("AVAILABLE MEMORY")
150  INITIALIZE ":,0,1",0               ! Try to reclaim unit 1 (can't).
160  PRINT SYSTEM$("AVAILABLE MEMORY")
170  INITIALIZE ":,0,2",4               ! Try to reclaim unit 2 (can).
180  PRINT SYSTEM$("AVAILABLE MEMORY")
190  INITIALIZE ":,0,0",4               ! Try to reclaim unit 0 (can).
200  PRINT SYSTEM$("AVAILABLE MEMORY")
210  END
```

Typical results:

```
434428
431314
431314
433390
434428
```

You can re-initialize a removed memory volume in its trapped space provided the newly allocated space is no larger than the original space that was allocated. Otherwise, new space will be allocated for it. (This happens even if enough trapped space exists for the new size.)

A result of being able to remove memory volumes is you are able to reclaim memory volume space without losing special typing aids or other customizations which SCRATCH A would undo.

9

I/O in the HP-UX Environment (BASIC/UX only)

This chapter covers some I/O topics specific to BASIC/UX running under HP-UX.

Using Interfaces in the HP-UX Environment

This section explains why you would want to lock an interface to an HP-UX process. It also covers using the BASIC/UX burst I/O mode.

Locking an Interface to a Process

In a multi-user environment, interface cards are usually accessible to several users. BASIC/UX supports this sharing by making no attempt to guarantee exclusive access to an interface *unless it is directed to do so*. This allows you to access instruments, for instance, on an HP-IB bus that is shared with other peripherals. Although this is not a recommended configuration, it is allowed.

BASIC/UX provides interface locking to support exclusive access to an interface. When an interface is locked to a process, all other processes are prevented from using that interface. For instance, this feature can prevent the loss of important data while a process is taking measurements from an instrument by keeping other users or processes from using the same interface.

Interface locking is enabled and disabled by using pseudo-register 255 and the interface's select code. For example:

`CONTROL 7,255;1`	*Enables HP-IB interface locking.*
`CONTROL 7,255;0`	*Disables HP-IB interface locking.*

In order to be a "good citizen" on a multi-user system, you should unlock an interface after you no longer need to have it locked.

Note that attempting to lock an HP-IB connected to a system disk will result in an error.

In addition, attempting to lock an interface that is already locked to another process will cause a program to suspend execution until:

- The interface is unlocked (by the other process to which it is currently locked).
- A timeout occurs.
- You press Reset or `Clr I/O`.

Using the Burst I/O Mode (Series 300/400 Only)

The default mode of HP-UX I/O transactions requires many time consuming HP-UX system calls to transmit or receive data to the destination.

Another method, "burst I/O", maps the interface into your "user address space", thereby bypassing the memory buffer. This direct-write method decreases the number of calls to HP-UX I/O system routines, which establishes a short, highly tuned path for performing I/O operations. The interface is also implicitly locked when burst mode is enabled (see above explanation of interface locking).

Burst I/O provides the fastest I/O performance available with BASIC/UX for the "smaller" I/O transactions that are typical of many instruments. For instance, an 8-byte ENTER operation is over an order of magnitude faster when burst mode is enabled. For larger I/O operations, of more than 4 000 bytes for example, burst mode becomes increasingly slower than the default (buffered or DMA) I/O modes.

Burst I/O is enabled and disabled by using pseudo-register 255 and the interface's select code. For example:

`CONTROL 7,255;3`	*Enables HP-IB interface burst I/O.*
`CONTROL 7,255;0`	*Disables HP-IB interface burst I/O.*

In addition, attempting to use burst mode with an interface that is already locked to another process will cause a program to suspend execution until:

- The interface is unlocked (by the other process to which it is currently locked).
- A timeout occurs.
- You press Reset or `Clr I/O`.

Note also that you cannot set up an ON TIMEOUT for an interface when using burst mode.

Burst I/O Mode Memory Allocation Failure (Error 811) (Series 300/400 Only)

If you do not have enough memory for the program you are running, you will receive the following error message:

```
ERROR 811 Memory allocation failed
```

Solutions for the Error 811 Memory Allocation Problem

If you get `ERROR 811` when using the burst I/O mode, you may need to increase the size of your heap space. Information for increasing the heap space is provided in a subsequent section. For information on the burst I/O mode, refer to the earlier section titled "Using the Burst I/O Mode".

If `ERROR 811` should occur when you are not using the burst I/O mode, you need to do one of the following to correct the problem:

- move the Starbase shared memory segment (`SB_DISPLAY_ADDR`)
- shrink the BASIC/UX workspace size (`RMB_SHMEM_ADDR`)
- reconfigure the kernel to allow a larger data segment and larger maximum shared memory address.

Increasing the Heap Space

To increase your heap space, add the following variable to the BASIC/UX environment file called `/usr/lib/rmb/rmbrc` (or `~/.rmbrc`):

```
HEAP_PREALLOC= additional_heap_space
```

If the *additional_heap_space* given in bytes is zero (the default value), then no additional heap space is allocated; however, if it is non-zero then the amount of heap space specified is preallocated.

Some Heap-consuming BASIC/UX Operations

Several heap-consuming BASIC operations are listed below, as well as suggested amounts of heap space to add for each one if the need arises:

CREATE WINDOW 17Kbytes for the default window and buffer sizes

GLOAD/GSTORE *width* × *height* of "from" device (if given) or "PLOTTER IS" device (if no parameters)

INITIALIZE *memory volume number of sectors* × 256 bytes

CSUBS size of stored CSUB file

BPLOT *width* × *height* for given parameters (plus size of stored CSUB)

GDUMP_R *width* × *height* of "from" device (plus size of stored CSUB)

DUMP GRAPHICS *width* × *height* of "from" device or `PLOTTER IS` device if no parameter is given

mass memory operations on HFS directories will at most use 20 Kbytes

opening SRM file each open file uses 48 bytes

Using HP-UX Pipes

This section explains what pipes are and how to use them in HP-UX. Topics covered are:

- An overview of shells and processes
- What are pipes?
- Connecting processes with pipes
- Using the `tee` command with pipes
- Using pipes with BASIC/UX and BASIC/WS on SRM/UX

An Overview of Shells and Processes

A **shell** is a command interpreter; that is, a shell interprets the text you type at the keyboard and directs the HP-UX operating system to take an appropriate action. Shells also serve as a programming language, allowing you to put commands into a file and execute the file like a program.

After a shell interprets a command line, HP-UX loads the named program into memory and begins running the program. When a program is loaded and running, it is called a **process**. When a process begins executing, HP-UX automatically opens three files for the process: standard input, standard output, and standard error. The following list is an explanation of the files that are opened up by a process:

Standard input is the place from which a program expects to read its input. By default, processes read standard input (`stdin`) from the keyboard. For example:

```
$ mail jam
Remember the meeting today a 4:00.      Standard input.
CTRL-D                                  End of standard input.
```

Standard output is the place the program writes its output. By default, processes write standard output (`stdout`) to the terminal screen. For example:

```
$ whoami
tim                                     Standard output.
```

Standard error is the place the program writes its error messages. By default, processes write standard errors (`stderr`) to the terminal screen. For example:

```
$ cta file_one
cta: not found                        Standard error.
```

What are Pipes?

HP-UX provides **named** and **unnamed** pipes for use in transferring data between processes. This section explains what named and unnamed pipes are.

Unnamed Pipes

The shell allows you to connect two or more processes so the standard output of one process is used as the standard input to another process. The connection that joins the processes is called a **pipe**. The use of **unnamed pipes** within a command line is called a **pipeline**. To pipe the output of one process into another, separate the commands in the command line with a vertical bar (|).

A pipe can link any two process as long as the first process writes its output to `stdout` and the second program reads its input from `stdin`.

The general syntax for a pipe is:

command1 | *command2*

where *command1* is the command whose standard output is redirected or "piped" to another command, and *command2* is the command whose standard input reads the previous command's output. Two or more commands can be combined together into a single pipeline. Each successive command has its output piped as input into the following command:

command1 | *command2* | ... | *commandN*

Named Pipes

This form of HP-UX pipe differs from an unnamed pipe in the following ways:

- It allows two or more otherwise unconnected processes to communicate with each other. For example, the following set of commands takes the output of *command_1* and writes it to *named_pipe*. Then *command_2* uses *named_pipe* for its input.

  ```
  command_1 >> named_pipe &

  command_2 < named_pipe
  ```

  Note that, in order to execute two processes in the same shell, the first process must be executed in the background as seen above (an "&" follows the command). Both of these process could also have been executed from different shells running in two different windows of the X Window system. For example, this command could be executed in window 1:

  ```
  command_1 >> named_pipe
  ```

  while this command is executed in window 2:

  ```
  command_2 < named_pipe
  ```

- In general it should not be used to combine the outputs and inputs of more than two processes because undetermined results may occur.

A named pipe differs from a file in the following ways:

- It is created using the `/etc/mknod` command. For example, executing the following command:

  ```
  /etc/mknod file_name p (Return)
  ```

  creates a named pipe called *file_name*. You can change the read and write permissions of the named_pipe with the `chmod` command.

- It can be only written to and read from serially but files can be written to and read from randomly or serially.
- Once the data is read, it is no longer kept in memory (or in the file system).

Why would you want to use a named pipe instead of a file? Once data is written by a process to a named pipe and it has been read by another process, that data is removed from memory. This eliminates the need for temporary files that clutter up the file system and use up system memory. Note that the same thing is true for unnamed pipes.

Connecting Processes with Pipes

You can use pipes whenever the output of one command is needed by a second command. In the following example, output from the `who` command is stored in the file `newwhoison`. Then, the `newwhoison` file is used as input to the `wc` command:

```
who > newwhoison
wc -l newwhoison
```

With an unnamed pipe, these two commands become one:

```
who | wc -l
```

The above task can also be performed using a named pipe. For example, the following set of commands takes the output of the `who` command and writes it to the named pipe called `pipe`. Then the `wc -l` command uses the named pipe (`pipe`) for its input to determine how many users are logged onto the system.

```
who >> pipe &
wc -l < pipe
```

Both of these processes can be executed from different shells running in two different windows of the X Window system. For example:

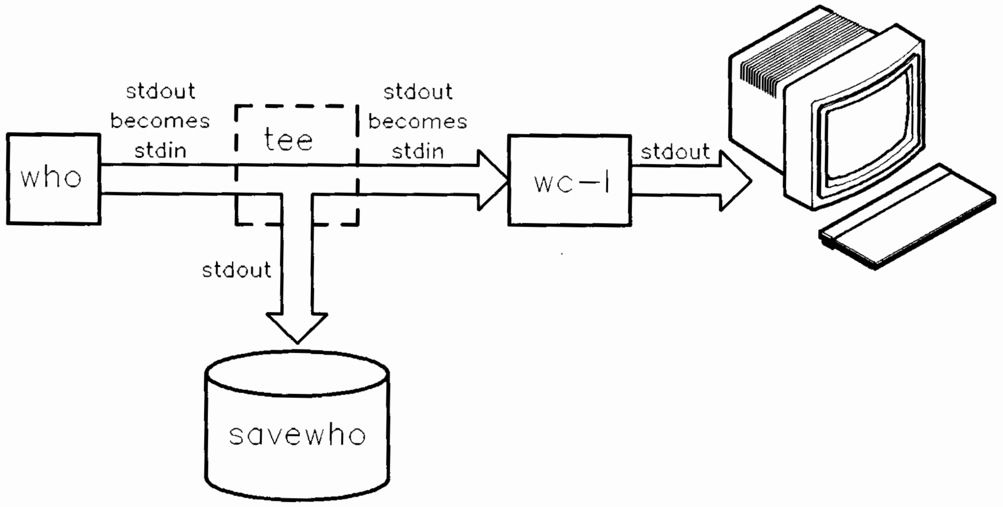
`who >> pipe`	*executed in window one*
`wc -l pipe`	*executed in window two*

Using the tee Command with Pipes

The `tee` command allows you to divert a copy of the data passing between commands to a file without changing the way the pipeline functions. The following example uses the `who` command to determine who is on the system. The output from `who` is piped into the `tee` command. The `tee` commands saves a copy of the output in the file **savewho**, and passes the output on, without change, to the `wc` command:

```
$ who | tee savewho | wc -l
      4
$ cat savewho
pat        console      Oct  9 08:50
terry      tty01        Oct  9 11:57
kim        tty02        Oct  9 08:13
kelly      tty04        Oct  9 10:04
$
```

The following figure illustrates what is happening in this example.



Standard Input and Output with Pipes and tee Command

For More Information ...

HP-UX provides many programs that are extremely useful in pipelines. These programs are called **filter** programs because they accept text as input, transform the text in some way, and produce text as output. Examples of filter commands include:

```
adjust    cut    head    pr    sed    spell
cat       grep   more    rev   sort   tail
```

For information on these commands, refer to the *HP-UX Reference Vol. 1:Section 1* manual.

Using Pipes with BASIC/UX and BASIC/WS on SRM/UX

Pipes can be used by BASIC/UX commands to send their output to an HP-UX command or to receive their input from an HP-UX command. The methods used to do this are:

- Inbound pipes
- Outbound pipes
- Named pipes.

Note HP BASIC/WS on SRM/UX uses named pipes only.

Named and unnamed pipes in BASIC/UX can be assigned to I/O path names. For example:

```
ASSIGN @Unnamed_pipe TO "| lp"
```

In this example the unnamed pipe "`| lp`" is assigned to the I/O path name called `@Unnamed_pipe`. You can also assign named and unnamed pipes directly with these BASIC statements:

PRINTER IS	PRINTALL IS
PLOTTER IS	DUMP DEVICE IS

For example, the following statement assigns the printing device to be the system's printer spooler:

```
PRINTER IS "| lp"
```

Inbound Pipes

This type of pipe takes the standard output of an HP-UX command and uses it as input to a BASIC command. Some examples of these BASIC commands are:

ENTER

TRANSFER

The following example takes the output of the HP-UX `date` command and uses it as input to the BASIC command ENTER. It then prints the result.

```
100   DIM Date$[40]          ! Dimensions the string called Date$.
110   ASSIGN @Date TO "date |"  ! Assigns the I/O path name @Date
120                             ! to the inbound pipe.
130   ENTER @Date USING "K";Date$ ! Inputs data from the HP-UX
140                               ! date command into the string
150                               ! Date$.
160   PRINT Date$                 ! Prints the contents of Date$.
170   END
```

Outbound Pipes

This type of pipe takes the standard output of a BASIC command and uses it as input to an HP-UX command. Examples of BASIC commands used for this method of piping are:

OUTPUT	TRANSFER
PRINT	PRINTALL
DUMP	

The following example takes the output of the OUTPUT command and sends it to the HP-UX system's printer spooler.

```
100   ASSIGN @Spooler TO "| lp"  ! Assigns the outbound pipe
110                              ! to the I/O path name @Spooler.
120   OUTPUT @Spooler USING "K";"This is a test of outbound pipes."
130   END
```

Named Pipes

These pipes take the output of a process and use it as the input to another process. For example, you could have BASIC/UX running in two different X Windows and each BASIC/UX process could have a different BASIC program running. The program in the first BASIC/UX window could be outputting data to a named pipe and the program in the second BASIC/UX window could be inputing data from the named pipe. The following BASIC programs running in different BASIC/UX windows perform this task. First create a named pipe by executing the following HP-UX command:

```
/etc/mknod pipe p  (Return)
```

This command can be executed in a directory and X Window of your choosing. You will also need to create a second X Window. For information on X Windows and how to create them, read the *Using the X Window System* manual. In each X Window start the BASIC/UX process, by executing the following command:

```
rmb  (Return)
```

Once you have BASIC/UX running in both of these X Windows, you can enter the following programs and run them. The first program outputs the numbers 1 through 20 to the named pipe called pipe and prints them. The second program inputs the values read from the named pipe and prints them.

Program Running in the First Window

```
100   ASSIGN @Pipe TO "pipe" ! Assigns the named pipe to the
110                          ! the I/O path name @Pipe.
120   FOR I=1 TO 20
130     OUTPUT @Pipe;I       ! Outputs the numbers 1 through
140                          ! 20 to the named pipe called "pipe."
150     PRINT I              ! Prints the values of the variable I.
160   END
```

Program Running in the Second Window

```
100   ASSIGN @Pipe TO "pipe" ! Assigns the named pipe to the
110                          ! the I/O path name @Pipe.
120   LOOP
130     ENTER @Pipe;Value    ! Inputs the numbers 1 through
140     PRINT Value          ! 20 from the named pipe called "pipe."
150     EXIT IF Value=20
160   END LOOP
170   END
```

This ability to move data between the computer and all of its resources with the same statements is a very powerful capability of the computer's BASIC language.

Before briefly discussing I/O paths to mass storage files, the following discussion will present some background information that will help you understand the rationale behind implementing the two data representations (ASCII and internal) used by the computer. The remainder of this chapter then presents several uses of this language structure.

Data-Representation Design Criteria

As you know, BASIC supports two general data representations: ASCII and internal. This discussion presents the rationale of their design.

The data representations used by BASIC were chosen according to the following criteria.

- to maximize the rate at which computations can be made
- to maximize the rate at which programs can move data between computer resources
- to minimize the amount of storage space required to store a given amount of data
- to be compatible with the data representation used by the resources with which the computer is to communicate

The *internal representations* implemented in BASIC are designed according to the *first three of the above criteria*. However, the last criterion must always be met if communication is to be achieved. If the resource uses the ASCII representation, this compatibility requirement takes precedence over the other design criteria. The *ASCII representation* fulfills this *last criterion* for most devices and for the computer operator. The first three criteria are further discussed in the following description of data representations used for mass storage files.

I/O Paths to Files

There are three types of *data files*: ASCII, BDAT, and HPUX.

- Only the ASCII data representation is used with ASCII files.
- But either the ASCII (FORMAT ON) or the internal (FORMAT OFF) representation can be used with BDAT and HPUX files.

I/O paths to files are briefly described in this section to further justify the internal data representations implemented with this system, and to preface the applications presented in the last section of this chapter.

BDAT Files

BDAT (BASIC Data) and HP-UX files have been designed with the first three of the preceding design criteria in mind. *Both numeric and string computations are much faster.* These internal data representations *generally allow much more data to be stored on a disk* because there is no storage overhead (for numeric items); that is, there are no "record headers" for numeric items.

The *transfer rates* for each data type has also been *increased*. Numeric output operations are always much faster because there is no time required for "formatting". Numeric enter operations are also faster because the system does not have to search for item- and statement-termination conditions.

In addition, I/O paths to BDAT and HPUX files can use either the ASCII (FORMAT ON) or the internal (FORMAT OFF) representation.

The following program shows a few of the features of BDAT files. (Examples of HP-UX files are shown in the "Porting and Sharing Files" chapter of *HP BASIC Porting and Globalization.*) The program first outputs an internal-form string (with FORMAT ON), and then enters the length header and string characters with FORMAT OFF.

```
100   OPTION BASE 1
110   DIM Length$[4],Data$[256],Int_form$[256]
120   !
130   ! Create a BDAT file (1 record; 256 bytes/record.)
140   ON ERROR GOTO Already_created
150   CREATE BDAT "B_file",1
160 Already_created: OFF ERROR
170   !
180   ! Use FORMAT ON during output.
190   ASSIGN @Io_path TO "B_file";FORMAT ON
200   !
210   Length$=CHR$(0)&CHR$(0)  ! Create length header.
    Length$=Length$&CHR$(0)&CHR$(252)
230   !
240   ! Generate 256-character string.
250   Data$="01234567"
260   FOR Doubling=1 TO 5
270      Data$=Data$&Data$
280   NEXT Doubling
290   ! Use only 1st 252 characters.
300   Data$=Data$[1,252]
310   !
```

```
320   ! Generate internal-form and output.
330   Int_form$=Length$&Data$
340   OUTPUT @Io_path;Int_form$;
350   ASSIGN @Io_path TO *
360   !
370   ! Use FORMAT OFF during enter (default).
380   ASSIGN @Io_path TO "B_file"
390   !
400   ! Enter and print data and # of characters.
410   ENTER Data$
420   PRINT LEN(Data$);"characters entered."
430   PRINT
440   PRINT Data$
450   ASSIGN @Io_path TO * ! Close I/O path.
460   !
470   END
```

ASCII Files

ASCII files are designed for interchangeability with other HP computer systems. This interchangeability imposes the restriction that the data must be represented with ASCII characters. Each data item sent to these files is a special case of FORMAT ON representation; *each item is preceded by a two-byte length header* (analogous to the internal form of string data). In order to maintain this compatibility, there are two additional restrictions placed on ASCII files:

- The FORMAT OFF attribute *cannot* be assigned to an ASCII file
- You cannot use OUTPUT..USING or ENTER..USING with an ASCII file.

The following program shows the I/O path name @Io_path being assigned to the ASCII file named ASC_FILE. Notice that the file name is in all upper-case letters; this is also a compatibility requirement when using this file with some other systems.

The program creates an ASCII file, and then outputs program lines to the file. The program then gets and runs this newly created program. (If you type in and run this program, be sure to save it on disk, because running the program will load the program it creates, destroying itself in the process.)

```
100   DIM Line$(1:3)[100]  ! Array to store program.
110   !
120   ! Create if not already on disk.
```

```
130   ON ERROR GOTO Already_exists
140   CREATE ASCII "ASC_FILE",1  ! 1 record.
150   Already_exists:  OFF ERROR
160   !
170   ASSIGN @Io_path TO "ASC_FILE"
180   STATUS @Io_path,6;Pointer
190   PRINT "Initially:    file pointer=";Pointer
200   PRINT
210   !
220   Line$(1)="100  PRINT ""New program.""  "
230   Line$(2)="110  BEEP"
240   Line$(3)="120  END"
250   !
260   OUTPUT @Io_path;Line$(*)
270   STATUS @Io_path,6;Pointer
280   PRINT "After OUTPUT: file pointer=";Pointer
290   PRINT
300   !
310   GET "ASC_FILE" ! Implicitly closes I/O path.
320   !
330   END
```

Data Representation Summary

The following table summarizes the control that programs have on the FORMAT attribute assigned to I/O paths.

Program Control of the FORMAT Attribute

Type of Resource	Default FORMAT Attribute Used	Can Default FORMAT Attribute Be Changed?
Devices	FORMAT ON	Yes (if an I/O path is used)[1]
BDAT files	FORMAT OFF	Yes
HPUX files	FORMAT OFF	Yes
ASCII files	FORMAT ON[2]	No
String variables	FORMAT ON	No
Buffers	FORMAT ON	Yes
Pipes	FORMAT ON	Yes

[1]FORMAT ON is *always* used whenever an OUTPUT..USING or ENTER..USING statement is used, regardless of the FORMAT attribute assigned to the I/O path.

[2]The data representation used with ASCII files is a special case of the FORMAT ON representation.

Applications of Unified I/O

This section describes two uses for the powerful, unified-I/O scheme of the BASIC language. Example 1 (in "Outputting Data to String Variables" section) contains further details and uses of I/O operations with string variables. Example 2 (in that section) involves using a disk file to simulate a device.

I/O Operations with String Variables

Chapter 14, "Introduction to I/O," in the *HP BASIC Programming Guide* briefly describes how string variables may be specified as the source or destination of data in I/O statements, but it describes neither the details nor many uses of these operations. This section describes both the details of and several uses of outputting data to and entering data from string variables.

Outputting Data to String Variables

When a string variable is specified as the destination of data in an OUTPUT statement, source items are evaluated individually and placed into the variable according to the free-field rules or the specified image, depending on which type of OUTPUT statement is used. Thus, item terminators may or may not be placed into the variable. The ASCII data representation is always used during outputs to string variables.

Characters are always placed into the variable beginning at the first position; no other position can be specified as the beginning position at which data will be placed. Thus, *random access of the information in string variables is not allowed* from OUTPUT and ENTER statements; all data must be accessed serially. For instance, if the characters "1234" are output to a string variable by one OUTPUT statement, and a subsequent OUTPUT statement outputs the characters "5678" to the same variable, the second output *does not* begin where the first one left off (i.e., at string position five). The second OUTPUT statement begins placing characters in position one, just as the first OUTPUT statement did, overwriting the data initially output to the variable by the first OUTPUT statement.

The string variable's length header (4 bytes) is updated and compared to the dimensioned length of the string as characters are output to the variable. If the string is filled before all items have been output, an error is reported; however, the string contains the first *n* characters output (where *n* is the dimensioned length of the string).

Example 1

The following program outputs string and numeric data items to a string variable and then calls a subprogram which displays each character, its decimal code, and its position within the variable.

```
100   ASSIGN @Crt TO 1  ! CRT is disp. device.
110   !
120   OUTPUT Str_var$;12,"AB",34
130   !
140   CALL Read_string(@Crt,Str_var$)
150   !
160   END
170   !
180   !
190 SUB Read_string(@Disp,Str_var$)
200     !
210     ! Table heading.
220     OUTPUT @Disp;"---------------------"
230     OUTPUT @Disp;"Character  Code  Pos."
240     OUTPUT @Disp;"---------  ----  ----"
250     Dsp_img$="2X,4A,5X,3D,2X,3D"
260     !
270     ! Now read the string's contents.
280   FOR Str_pos=1 TO LEN(Str_var$)
290         Code=NUM(Str_var$[Str_pos;1])
300       IF Code<32 THEN ! Don't disp. CTRL chars.
310           Char$="CTRL"
320       ELSE
330           Char$=Str_var$[Str_pos;1] ! Disp. char.
340       END IF
350       !
360       OUTPUT @Disp USING Dsp_img$;Char$,Code,Str_pos
370   NEXT Str_pos
380   !
390   ! Finish table.
400   OUTPUT @Disp;"---------------------"
410   OUTPUT @Disp ! Blank line.
420   !
430   SUBEND
```

```
---------------------
Character  Code  Pos.
---------  ----  ----
             32    1
1            49    2
2            50    3
,            44    4
A            65    5
B            66    6
CTRL         13    7
CTRL         10    8
             32    9
3            51   10
4            52   11
CTRL         13   12
CTRL         10   13
---------------------
```

Final Display

Outputting data to a string and then examining the string's contents is usually a more convenient method of examining output data streams than using a mass storage file. The preceding subprogram may facilitate the search for control characters, because they are not actually displayed, which could otherwise interfere with examining the data stream.

Example 2

The following example program shows how outputs to string variables can be used to reduce the overhead required in ASCII data files. The first method of outputting data to the file requires as much media space for overhead as for data storage, due to the two-byte length header that precedes each item sent to an ASCII file. The second method uses more computer memory, but uses only about half of the storage-media space required by the first method. The second method is also *the only way to format data sent to ASCII data files.*

```
100   PRINTER IS CRT
110   !
120   ! Create a file 1 record long (=256 bytes).
130   ON ERROR GOTO File_exists
140   CREATE ASCII "TABLE",1
150 File_exists:   OFF ERROR
```

```
160              !
170              !
180   ! First method outputs 64 items individually..
190   ASSIGN @Ascii TO "TABLE"
200   FOR Item=1 TO 64  ! Store 64 2-byte items.
210       OUTPUT @Ascii;CHR$(Item+31)&CHR$(64+RND*32)
220       STATUS @Ascii,5;Rec,Byte
230       DISP USING Image_1;Item,Rec,Byte
240   NEXT Item
ed amount=4.0in>
250 Image_1: IMAGE "Item ",DD,"  Record ",D,"  Byte ",3D
260   DISP
270   Bytes_used=256*(Rec-1)+Byte-1
280   PRINT Bytes_used;" bytes used with 1st method."
290   PRINT
300   PRINT
310   !
320   !
330   ! Second method consolidates items.
340   DIM Array$(1:64)[2],String$[128]
350   ASSIGN @Ascii TO "TABLE"
360   !
370   FOR Item=1 TO 64
380       Array$(Item)=CHR$(Item+31)&CHR$(64+RND*32)
390   NEXT Item
400   !
410   OUTPUT String$;Array$(*); ! Consolidate.
420   OUTPUT @Ascii;String$     ! OUTPUT as 1 item.
430   !
440   STATUS @Ascii,5;Rec,Byte
450   Bytes_used=256*(Rec-1)+Byte-1
460   PRINT Bytes_used;" bytes used with 2nd method."
470   !
480   END
```

The program shows many of the features of using ASCII files and string variables. The first method of outputting the data shows how the file pointer varies as data items are sent to the file. Note that the file pointer points to the *next* file position at which a subsequent byte will be placed. In this case, it is incremented by four by every OUTPUT statement (since each item is a two-byte quantity preceded by a two-byte length header).

The program could have used a BDAT file, which would have resulted in using slightly less disk-media space; however, using BDAT files usually saves much more disk space than would be saved in this example. The program does not show that *ASCII files cannot be accessed randomly*; this is one of the major differences between using ASCII and BDAT files.

Example

Outputs to string variables can also be used to generate the string representation of a number, rather than using the VAL$ function (or a user-defined function subprogram). The *main advantage* is that you can explicitly specify the number's image while still using only a single program line. The following program compares the string generated by the VAL$ function to that generated by outputting the number to a string variable.

```
100   X=12345678
110   !
120   PRINT VAL$(X)
130   !
140   OUTPUT Val$ USING "#,3D.E";X
150   PRINT Val$
160   !
170   END
```

Printed Results

```
1.2345678E+7
123.E+05
```

Entering Data From String Variables

Data are entered from string variables in much the same manner as output to the variable. All ENTER statements that use string variables as the data source interpret the data according to the FORMAT ON attribute. Data are read from the variable beginning at the first string position; if subsequent ENTER statements read characters from the variable, the read also begins at the first position. If more data are to be entered from the string than are contained in the string, an error is reported; however, all data entered into the destination variable(s) before the end of the string was encountered remain in the variable(s) after the error occurs.

When entering data from a string variable, BASIC keeps track of the number of characters taken from the variable and compares it to the string length.

Thus, *statement-termination* conditions are *not* required; the ENTER statement automatically terminates when the last character is read from the variable. However, *item* terminators are still required *if* the items are to be separated *and* the lengths of the items are not known. If the length of each item is known, an image can be used to separate the items.

Example

The following program shows an example of the need for *either* item terminators *or* length of each item. The first item was not properly terminated and caused the second item to not be recognized.

```
100   OUTPUT String$;"ABC123";  ! OUTPUT w/o CR/LF.
110   !
120   ! Now enter the data.
130   ON ERROR GOTO Try_again
140   !
150 First_try: !
160   ENTER String$;Str$,Num
170   OUTPUT 1;"First try results:"
180   OUTPUT 1;"Str$= ";Str$,"Num=";Num
190   BEEP     ! Report getting this far.
200   STOP
210   !
220 Try_again: OUTPUT 1;"Error";ERRN;" on 1st try"
230            OUTPUT 1;"STR$=";Str$,"Num=";Num
240            OUTPUT 1
250            OFF ERROR  ! The next one will work.
260            !
270   ENTER String$ USING "3A,3D";Str$,Num
280   OUTPUT 1;"Second try results:"
290   OUTPUT 1;"Str$= ";Str$,"Num=";Num
300   !
310   END
```

This technique is convenient when attempting to enter an unknown amount of data or when numeric and string items within incoming data are not terminated. The data can be entered into a string variable and then searched by using images.

Example

ENTERs from string variables can also be used to generate a number from ASCII numeric characters (a recognizable collection of decimal digits, decimal point, and exponent information), rather than using the VAL function. As with outputs to string variables, images can be used to interpret the data being entered.

```
30    Number$="Value= 43.5879E-13"
40    !
50    ENTER Number$;Value
60    PRINT "VALUE=";Value
70    END
```

Taking a Top-Down Approach

This application shows how the computer's BASIC-language structure may help simplify using a "top-down" programming approach. In this example, a simple algorithm is first designed and then expanded into a program in a general-to-specific, stepwise manner. The top-down approach shown here begins with the general steps and works toward the specific details of each step in an orderly fashion.

One of the first things you *should* do when programming computers is to *plan the procedure before actually coding any software*. At this point of the design process, you need to have a good understanding of both the problem and the requirements of the program. The general tasks that the program is to accomplish must be described before the order of the steps can be chosen. The following simple example goes through the steps of taking this top-down approach to solving the problem.

Sample Problem

Write a program to monitor the temperature of an experimental oven for one hour.

Step 1 *Verbally describe what the program must do in the most general terms.*

You may want to make a chart or draw a picture to help visualize what is required of the program. For example:

a. Initialize the monitoring equipment.
b. Start the timer and turn the oven on.
c. Begin monitoring oven temperature and measure it every minute thereafter for one hour.
d. Display the current oven temperature, and plot the temperatures vs. time on the CRT.

Step 2 *Verbally describe the algorithm. Again, try to keep the steps as general as possible.*

This process is often termed writing the "pseudo code". Pseudo code is merely a written description of the procedure that the computer will execute. The pseudo code can later be translated into BASIC code. For example:

```
a. Setup the equipment.
b. Set the oven temperature and turn it on.
c. Initialize the timer.
d. Perform the following tasks every minute for one hour.
   ■ Read the oven temperature.
   ■ Display the current temperature and elapsed time.
   ■ Plot the temperature on the CRT.
e. Turn the oven and equipment off.
f. Signal that the experiment is done.
```

Step 3 *Begin translating the algorithm into a BASIC program.*

The following program follows the general flow of the algorithm. As you become more fluent in a computer language, you may be able to write pseudo code that will translate more directly into the language. However, avoid the temptation to write the initial algorithm in the computer language, because writing the pseudo code is a *very important* step of this design approach!

```
100   ! This program: sets up measuring equipment,
110   ! turns an oven on, and initializes a timer.
120   ! The oven's temperature is measured every
130   ! minute thereafter for one hour. The temp.
140   ! readings are displayed and plotted on the
150   ! CRT.
160   !
170   Rdgs_interval=60  ! 60 seconds between readings.
180   Test_length=60    ! Run test for 60 minutes.
190   !
200   CALL Equip_setup
210   CALL Set_temp
220   GOSUB Start_timer
230   !
240 Keep_monitoring: ! Main loop.
250                 !
260   GOSUB Timer
270   !
280   IF Seconds<=Rdgs_interval THEN
290       GOTO Keep_monitoring
300   ELSE
310       Minutes=Minutes+1
320       CALL Read_temp
330       CALL Plot_temp
340   END IF
350   !
360   !
370   IF Minutes<Test_length THEN
380       GOTO Keep_monitoring
390   ELSE
400       CALL Off_equip
410       PRINT "End of experiment"
420   END IF
430   !
440   STOP
450   !
460   !
470   ! First the subroutines.
480   !
490 Start_timer: Init_time=TIMEDATE
500              PRINT "Timer initialized."
510              PRINT
520              PRINT
530              RETURN
540              !
```

```
550 Timer: !
560        Seconds=TIMEDATE-Minutes*60-Init_time
570        DISP USING Time_image;Minutes,Seconds
580 Time_image: IMAGE "Time: ",DD," min  ",DD.D," sec"
590        RETURN
600        !
610   END
620   !
630   !
640   ! Now the subprograms.
650   !
660   SUB Equip_setup
670      PRINT "Equipment setup."
680      SUBEND
690      !
700   SUB Set_temp
710      PRINT "Oven temperature set."
720      SUBEND
730      !
740   SUB Read_temp
750      PRINT "Temp.= xx degrees F  ";
760      SUBEND
770      !
780   SUB Plot_temp
790      PRINT "(plotted)."
800      PRINT
810      SUBEND
820      !
830   SUB Off_equip
840      PRINT
850      PRINT "Equipment shut down."
860      PRINT
870      SUBEND
```

At this point, you should run the program to verify that the general program steps are being executed in the desired sequence. If not, keep refining the program flow until all steps are executed in the proper sequence. This is also a very important step of your design process; the sooner you can verify the flow of the main program the better. This approach also relieves you of having to set up and perform the actual experiment as the first test of the program.

Notice also that some of the program steps use CALLs while others use GOSUBs. The general convention used in this example is that subprograms are used only when a program step is to be expanded later. GOSUBs are used when the routine called will probably not need further refinement. As the subprograms are expanded and refined, each can be separately stored and loaded from disk files, as shown in the next step.

Step 4 *After the correct order of the steps has been verified, you can begin programming and verifying the details of each step (known as stepwise refinement).*

The computer features a mechanism by which the process of expanding each step can be simplified. With it, each subprogram can be expanded and refined individually and then stored separately in a disk file. This facilitates the use of the top-down approach. Each subprogram can also be tested separately, if desired.

In order to use this mechanism, first save or store the main program; for instance, execute:

```
SAVE "MAIN1"
```

Then, isolate the subprogram by deleting all other program lines in memory. In this case, executing:

```
DEL 10,650
```

and

```
DEL 700,900
```

would delete the lines which are not part of the "Equip_setup" subprogram currently in memory.

```
660   SUB Equip_setup
670       PRINT "Equipment setup."
680   SUBEND
690   !
```

At this point, two steps can be taken:

- Write the temperature-measuring device's initialization routine.
- Write a test routine that simulates the device by returning a known set of data.

The "Equip_setup" subprogram might be expanded as follows to create a disk file and fill it with a known set of temperature readings so that the program can be tested without having to write, verify, and refine the routine that will set up the temperature-measuring device. In fact, you don't even need the device at this point.

```
100  CALL Equip_setup(@Temp_meter,Temp)
110  END
120  !
130  SUB Equip_setup(@Temp_meter,Temp)
140    !
150    ! This subroutine will set up a BDAT file as
160    ! be used to simulate a temperature-measuring
170    ! device. Refine to set up the actual
180    ! equipment later.
190    !
200    ON ERROR GOTO Already
210    CREATE BDAT "Temp_rdgs",1
220    !
230    ! Output fictitious readings.
240    ASSIGN @Temp_meter TO "Temp_rdgs"
250    FOR Reading=1 TO 60
260        OUTPUT @Temp_meter;Reading+70
270    NEXT Reading
280    ASSIGN @Temp_meter TO * ! Reset pointer.
290    !
300 Already: OFF ERROR
310    !
320    ASSIGN @Temp_meter TO "Temp_rdgs"
330    !
340    PRINT "Equipment setup."
350  SUBEND
```

Notice that two pass parameters have been added to the formal parameter list. These parameters allow the main program (and and subprograms to which these parameters are passed) to access this I/O path and variable. The CALL statements in the main program must be changed accordingly before the main program can be run with these subprograms. These parameters can also be passed to the subprograms by declaring them in variable common (that is, by including the appropriate COM statements).

After the subprogram has been expanded, tested, and refined, you should store it in a file with the STORE statement (not SAVE). For instance, execute:

```
STORE "SETUP1"
```

When the main program is to be tested again, the "Equip_setup" subprogram can be loaded back into memory by executing:

```
LOADSUB ALL FROM "SETUP1"
```

Since this subprogram names an I/O path which is to be used to simulate the temperature-measuring device, the "Read_temp" subprogram can also be expanded at this point. The "Read_temp" subprogram only needs to enter a reading from the measuring device (in this case, the disk file which has been set up to simulate the temperature-measuring device.) The following program shows how this subprogram might be expanded.

```
740  SUB Read_temp (@Temp_meter,Temp)
741      ENTER @Temp_meter;Temp
750      PRINT "Temp. =";Temp;" degrees F. "
760  SUBEND
```

This subprogram can also be stored in a disk file by executing:

```
STORE "READ_T1"
```

Now that both of the expanded subprograms have been stored, the main program can be retrieved and modified as necessary. Execute:

```
LOAD "MAIN1"
```

or

```
GET  "MAIN1"
```

Add the pass parameters to the appropriate CALL statements (lines 200 and 320). Since the main program still contains the initial versions of the expanded subprograms, these two subprograms should be deleted. Executing these two statements:

```
DELSUB "Equip_setup"
```

and

```
DELSUB "Read_temp"
```

will delete only these two subprograms and leave the rest of the program intact.

Now that the main program has been modified to CALL the expanded/refined subprograms, you may want to store (or save) a copy of the program on the disk. This will relieve you of the effort of deleting the old subprograms from the main program every time it is retrieved. Execute:

```
STORE "MAIN2
```

or

```
SAVE "MAIN2"
```

Now load the subprograms into memory by executing:

```
LOADSUB ALL FROM "SETUP1"
```

and

```
LOADSUB ALL FROM "READ_T1"
```

Running the program first "sets up" the device simulation and then accesses the file as it would access the actual temperature-measuring device.

Conclusion

As you can see, you can use this approach very easily with HP BASIC. You *do not have to revise* the "Read_temp" subprogram to access the real device. You only need to change "Equip_setup" to assign the I/O path name "@Temp_meter" to the real device. This unified I/O scheme makes this system very powerful and reduces "throw away" code because it uses a "top down" approach.

A
BASIC Driver Names and Peripheral Configurations (Series 200/300/400 Only)

Numerous interface cards and peripherals are available for the HP 9000 Series 200/300/400 computers. This appendix gives BASIC software configuration information for several types of interfaces and peripherals. All of this information is applicable to BASIC/WS. However, there are several exceptions for BASIC/UX, and these are noted throughout this appendix.

For further information, refer to the following sources:

- For hardware configuration information (switch settings, etc.), refer to the owner's documentation provided with your specific peripheral.
- If you are using BASIC/UX, refer to your HP-UX documentation for information about configuring peripherals through HP-UX.
- If you are using BASIC/DOS, refer to *Installing and Using HP BASIC/DOS* for information about configuring PC peripherals through the HP Measurement Coprocessor.

Interface Select Codes and Drivers

In order to access an interface from HP BASIC, you will need to use the correct select code in an HP BASIC statement (for example, OUTPUT). For BASIC/WS, you will also need to load the appropriate driver (binary). (For BASIC/UX 300/400, you won't need to load any binaries since BASIC/UX 300/400 is installed as a complete system.)

Note For HP BASIC/UX 700, all I/O interface select codes use the SICL library.

Loading Drivers (BASIC/WS Only)

To access an interface, first make sure that the interface driver is loaded and that its select code is listed during the system boot. For example, you may see a message such as `HP PARALLEL at 23` in the boot screen.

To load the driver, use the LOAD BIN command. For example, insert the appropriate system disk in your flexible disk drive, and execute:

```
LOAD BIN "PLLEL" (Return)
```

to load the parallel interface driver.

(For more information about the BASIC boot screen, refer to your *Installing and Maintaining HP BASIC* manual.)

Default Select Codes and Drivers

The following table lists the factory default select codes for several Series 200/300/400 interfaces. (Some of these select codes are usable only for BASIC/WS, as noted.) The name of the BASIC/WS driver (binary) is also given for each interface.

Table A-1. Default Select Codes and Drivers

Interface Name	Select Code	Driver Name (BASIC/WS Only)
HP 98253A EPROM Programmer Interface (BASIC/WS only)	27	EPROM
HP 98287A Graphics Display Station Interface (see note following table).	25 & 31	CRTB, GRAPH, and GRAPHX
HP 98546A Display Compatibility Interface (CRTB is needed if using a bit-mapped display)	1 or 3	CRTA, CRTB, GRAPH, and GRAPHX
HP 98622A GPIO Interface	12	GPIO
HP 98623A BCD Interface (BASIC/WS only)	11	BCD
HP 98624A HP-IB Interface	8	HPIB

Table A-1. Default Select Codes and Drivers (continued)

Interface Name	Select Code	Driver Name (BASIC/WS Only)
HP 98625A/B Disk Interface	14	FHPIB
HP 98626A RS-232C Interface	9	SERIAL
HP 98627A Color Output Interface	28	GRAPH and GRAPHX
HP 98628A Datacomm Interface	20	DCOMM
HP 98629A and 50961A SRM Interfaces	21	DCOMM and SRM
HP 98633A Multiprogrammer Interface	29	-
HP 98640A Analog Input Card	18	-
HP 98643A LAN Interface	21*	LAN
HP 98643A LAN Interface, using SRM/UX	21*	LAN and SRM
HP 98644A Asynchronous Serial Interface	9	SERIAL
HP 98695A IBM 3270 Coax Interface	22	-
HP 98265A SCSI Interface	14*	SCSI
HP 98658A SCSI Interface	14*	SCSI
Built-in SCSI interface (Model 340)	28*	SCSI
Built-in SCSI interface (Models 345, 362, 375, 380, 382)	14*	SCSI
Built-in HP Parallel interface	23*	PLLEL

* This select code is not supported by BASIC/UX. However, the interface can be accessed through HP-UX resources. (Refer to your HP-UX documentation.)

Note The HP 98287A Graphics Display Station Interface is the interface card for the HP 98700 Graphics Display Station. The default select codes for the HP 98287A are 25 for the graphics interface and 31 for the HP-HIL interface. BASIC does not support the HP-HIL interface, so leave its select code set to 31.

If you are using the HP 98287A in "external addressing" mode, leave the display select code at 25. If you have no other "internal" display, you can also use select code 1 to address the HP 98287A.

If you are using the HP 98287A in "internal addressing" mode, you can use select codes 1 and 6 to address the HP 98287A.

The HP 98562-66530 Human (System) Interface board provides four interfaces (HP-IB, RS-232C, Disk, and LAN) on one board. The following table gives the default select codes and BASIC/WS drivers.

Table A-2. HP 98562-66530 Select Codes and Drivers

Interface Name	Select Code	Driver Name (BASIC/WS Only)
Built-in HP-IB	7	HPIB
Built-in RS-232C (HP 98644)	9	SERIAL
Built-in Disk (HP 98625B)	14	FHPIB
Built-in LAN (HP 98643)	21*	LAN
Built-in LAN (HP 98643), using SRM/UX	21*	LAN and SRM

* This select code is not supported by BASIC/UX. However, the interface can be accessed through HP-UX resources. (Refer to your HP-UX documentation.)

Mass Storage Configurations

The following tables give configurations for several mass storage devices. The first table lists the MSVS (Mass Storage Volume Specifier), BASIC/WS driver name, and interleave factor for several devices connected to the HP 98625 Disk Interface (select code 14).

Note For HP BASIC/UX 700, all hard disks and magneto-optical drives go through SCSI drivers.

Also, for HP BASIC/UX 700, all SCSI floppy drives use the `SCSIfloppy` driver.

Note In each MSVS in the table, "n" indicates an address value in the range 0–7.

For BASIC/UX 300/400, fixed disk volumes are normally configured as HFS volumes and are accessed through the HP-UX file system drivers. Refer to your HP-UX documentation for further information. To access LIF flexible disk volumes from BASIC/UX 300/400, you can either use the MSVS configurations listed in the table, or you can use the HP-UX drivers.

Table A-3.
Mass Storage Configurations (HP 98625 Disk Interface)

Device Name	MSVS	Driver Name (BASIC/WS only)	Interleave Factor
HP 7907A - Fixed Disk	":,140n"	CS80	1
HP 7907A - Removable Disk	":,140n,1"	CS80	1
HP 7911/12/14 - Disk	":,140n"	CS80	1
HP 7911/12/14 - Tape	":,140n,1"	CS80	1
HP 7936H/37H	":,140n"	CS80	1
HP 7941/42/45/46 - Disk	":,140n"	CS80	1
HP 7942/46 - Tape	":,140n,1"	CS80	1
HP 7957A/58A	":,140n"	CS80	1

Table A-3.
Mass Storage Configurations (HP 98625 Disk Interface)
(continued)

Device Name	MSVS	Driver Name (BASIC/WS only)	Interleave Factor
HP 9133/34 - 1st fixed disk volume	":,140n"	CS80	3
HP 9133/34 - 2nd fixed disk volume	":,140n,0,1"	CS80	3
HP 9133/34 - 3rd fixed disk volume	":,140n,0,2"	CS80	3
HP 9133/34 - 4th fixed disk volume	":,140n,0,3"	CS80	3
HP 9133/34 - 5th fixed disk volume	":,140n,0,4"	CS80	3
HP 9133/34 - 6th fixed disk volume	":,140n,0,5"	CS80	3
HP 9133/34 - 7th fixed disk volume	":,140n,0,6"	CS80	3
HP 9133/34 - 8th fixed disk volume	":,140n,0,7"	CS80	3
HP 9133/34 - flexible	":,140n,1"	CS80	2
HP 9144A	":,140n"	CS80	1
HP 9153/54 - 1st fixed disk volume	":,140n"	CS80	1
HP 9153/54 - 2nd fixed disk volume	":,140n,0,1"	CS80	1
HP 9153/54 - 3rd fixed disk volume	":,140n,0,2"	CS80	1
HP 9153/54 - 4th fixed disk volume	":,140n,0,3"	CS80	1
HP 9153/54 - 5th fixed disk volume	":,140n,0,4"	CS80	1
HP 9153/54 - 6th fixed disk volume	":,140n,0,5"	CS80	1
HP 9153/54 - 7th fixed disk volume	":,140n,0,6"	CS80	1
HP 9153/54 - 8th fixed disk volume	":,140n,0,7"	CS80	1
HP 9153/54 - flexible	":,140n,1"	CS80	2
HP 7957/58/59B or HP 7961/62/63B	":,140n"	CS80	1

The following table lists the MSVS (Mass Storage Volume Specifier), BASIC/WS driver name, and interleave factor for several devices connected to the built-in HP-IB interface at select code 7. If you are using the HP 98624A HP-IB Interface at select code 8, just change the select code in the MSVS (for example, ":,70n" becomes ":,80n").

Note

In each MSVS in the table, "n" indicates an address value in the range 0–7.

For BASIC/UX, fixed disk volumes are normally configured as HFS volumes and are accessed through the HP-UX file system drivers. Refer to your HP-UX documentation for further information. To access LIF flexible disk volumes from BASIC/UX, you can either use the MSVS configurations listed in the table, or you can use the HP-UX drivers.

Table A-4. Mass Storage Configurations (Built-In HP-IB Interface)

Device Name	MSVS	Driver Name (BASIC/WS Only)	Interleave Factor
HP 7907A - Fixed Disk	":,70n"	CS80	1
HP 7907A - Removable Disk	":,70n,1"	CS80	1
HP 7911/12/14 - Disk	":,70n"	CS80	1
HP 7911/12/14 - Tape	":,70n,1"	CS80	1
HP 7941/42/45/46 - Disk	":,70n"	CS80	1
HP 7942/46 - Tape	":,70n,1"	CS80	1
HP 7957A/58A	":,70n"	CS80	1
HP 9121 - left	":,70n"	DISC	2
HP 9121 - right	":,70n,1"	DISC	2
HP 9122 - left	":,70n"	CS80	2
HP 9122 - right	":,70n,1"	CS80	2
HP 9125S/27A	":,70n"	CS80	2

Table A-4.
Mass Storage Configurations (Built-In HP-IB Interface)
(continued)

Device Name	MSVS	Driver Name (BASIC/WS Only)	Interleave Factor
HP 9133/34 - 1st fixed disk volume	":,70n"	CS80	7
HP 9133/34 - 2nd fixed disk volume	":,70n,0,1"	CS80	7
HP 9133/34 - 3rd fixed disk volume	":,70n,0,2"	CS80	7
HP 9133/34 - 4th fixed disk volume	":,70n,0,3"	CS80	7
HP 9133/34 - 5th fixed disk volume	":,70n,0,4"	CS80	7
HP 9133/34 - 6th fixed disk volume	":,70n,0,5"	CS80	7
HP 9133/34 - 7th fixed disk volume	":,70n,0,6"	CS80	7
HP 9133/34 - 8th fixed disk volume	":,70n,0,7"	CS80	7
HP 9133/34 - flexible	":,70n,1"	CS80	2
HP 9144A	":,70n"	CS80	1
HP 9153/54 - 1st fixed disk volume	":,70n"	CS80	7
HP 9153/54 - 2nd fixed disk volume	":,70n,0,1"	CS80	7
HP 9153/54 - 3rd fixed disk volume	":,70n,0,2"	CS80	7
HP 9153/54 - 4th fixed disk volume	":,70n,0,3"	CS80	7
HP 9153/54 - 5th fixed disk volume	":,70n,0,4"	CS80	7
HP 9153/54 - 6th fixed disk volume	":,70n,0,5"	CS80	7
HP 9153/54 - 7th fixed disk volume	":,70n,0,6"	CS80	7
HP 9153/54 - 8th fixed disk volume	":,70n,0,7"	CS80	7
HP 9153/54 - flexible	":,70n,1"	CS80	2
HP 7957/58/59B or HP 7961/62/63B	":,70n"	CS80	1

The following table lists configuration information for SCSI mass storage devices:

Note In each MSVS in the table, "n" indicates an address value in the range 0–7.

For BASIC/UX, SCSI disk volumes are accessed through the HP-UX file system drivers. Refer to your HP-UX documentation for further information.

Table A-5. SCSI Mass Storage Configurations (BASIC/WS Only)

Device Name	MSVS	Driver Name	Interleave Factor
HP 9000, Model 340 Internal Fixed Disk	":,280n"	SCSI	1
HP 9000, Model 345/375/380 Internal Fixed Disk	":,140n"	SCSI	1
HP 9000, Model 362/382 Internal Fixed Disk	":,140n"	SCSI	1
HP 9000, Model 362/382 Internal Flexible Disk	":,140n"	SCSI	1
HP 7957S, 7958S, and 7959S Fixed Disk	":,140n"	SCSI	1
HP 6000: 330S, 660S - Fixed Disk	":,140n"	SCSI	1
HP 6000: 330S, 660S - Rewritable Optical Disk	":,140n"	SCSI	n.a.
HP 6300: 650/A - Rewritable Optical Disk	":,140n"	SCSI	n.a.

The following table lists configurations for the HP 98255A EPROM Memory Card and the HP 98259A Bubble Memory Card (not supported by BASIC/UX).

Table A-6.
EPROM/Bubble Memory Configurations (BASIC/WS Only)

Device Name	MSVS	Driver Name	Interleave Factor
HP 98255, lowest address	":EPROM"	EPROM	n.a.
HP 98255, 2nd lowest address	":EPROM,0,1"	EPROM	n.a.
HP 98255, 3rd lowest address	":EPROM,0,2"	EPROM	n.a.
HP 98259	":,30"	BUBBLE	n.a.

Printer Configurations

HP BASIC supports many printers, too numerous for a complete listing here. For information about support of a particular printer, contact your Hewlett-Packard representative.

Note For HP BASIC/UX 700, Hewlett-Packard recommends that you access all printers through the `lp(1)` spooler system. Refer to your HP-UX documentation for details on the `lp(1)` spooler system.

In general, if a printer is supported, the only configuration question is what type of interface it uses. That is, is it an HP-IB printer, an RS-232C serial printer, or a parallel printer. The following table shows the software configurations. (The HP-IB device selectors include an "n" that indicates an HP-IB address value in the range 0–7.)

Table A-7. Printer Device Selectors and Drivers

Interface Name	Device Selector	Driver Name (BASIC/WS Only)
Printer connected to built-in HP-IB	70n	HPIB
Printer connected to HP 98624A HP-IB	80n	HPIB
Printer connected to RS-232C Interface	9	SERIAL
Printer connected to Parallel interface	23*	PLLEL

* This select code is not supported by BASIC/UX. However, the interface can be accessed through HP-UX resources. (Refer to your HP-UX documentation.)

Note For BASIC/UX, you can access printers either through device selectors, as indicated in the table—or, in most cases, through the HP-UX spooler. (Refer to your HP-UX documentation.)

Plotter Configurations

HP BASIC supports many plotters that use the HPGL plotter-control language. Other HPGL graphics devices, such as the HP 9111A Graphics Tablet, are also supported. For information about support of a particular plotter, refer to your Hewlett-Packard representative.

Note For HP BASIC/UX 700, Hewlett-Packard recommends that you access all plotters through the `lp(1)` spooler system. Refer to your HP-UX documentation for details on the `lp(1)` spooler system.

The following table shows the software configuration for an HPGL device connected to an HP-IB interface. (The device selectors include an "n" that indicates an HP-IB address value in the range 0–7.)

Table A-8. HPGL Device Selectors and Drivers

Interface Name	Device Selector	Driver Name (BASIC/WS Only)
HPGL device connected to built-in HP-IB	70n	HPGL
HPGL device connected to HP 98624A HP-IB	80n	HPGL

Note For BASIC/UX, you can access plotters either through device selectors, as indicated in the table, or through the HP-UX spooler. (Refer to your HP-UX documentation.)

Index

HEWLETT®
PACKARD

98616-90204