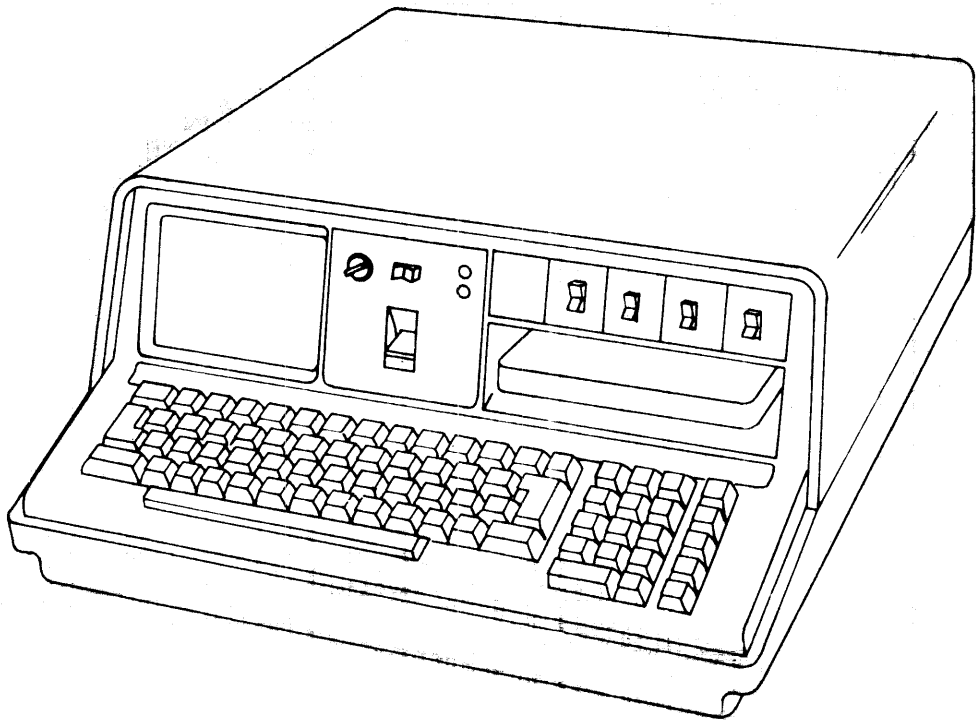
IBM

5110

IBM 5110
APL User's Guide



IBM 5110
APL User's Guide

SA21-9302-1

Second Edition (August 1978)

This is a major revision of, and obsoletes, SA21-9302-0. Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, be sure you have the latest edition and any technical newsletters.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Use this publication only for the purpose stated in the *Preface*.

Publications are not stocked at the address below. Requests for copies of IBM publications and for technical information about the system should be made to your IBM representative or to the branch office serving your locality.

This publication could contain technical inaccuracies or typographical errors. Use the Reader's Comment Form at the back of this publication to make comments about this publication. If the form has been removed, address your comments to IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901. Comments become the property of IBM.

Preface

This manual gives you conceptual information about using the 5110 with the APL language. Before using this manual, you should understand the information in the *IBM 5110 APL Introduction*, SA21-9301, especially:

- How to enter data from the keyboard
- How APL functions work with one or two arguments
- How to create your own user-defined functions using the built-in APL functions

This manual is intended to be used with the *IBM 5110 APL Reference Manual*, SA21-9303; that is, this manual gives you information about using the 5110 system with the APL language for information processing. The major topics are:

- Computer concepts for data processsing (Chapter 1)
- An approach to breaking your application into small parts to make programming easier (Chapter 2)
- Elements of the APL language and example APL user-defined functions used for information processing (Chapter 3)
- Controlling the information in the active workspace (Chapter 4)
- Using uppercase and lowercase characters, scrolling the display screen up or down, and sounding the audible alarm (Chapter 5)
- Creating, using, and maintaining your library (Chapter 6)
- Using the tape or diskette storage media (Chapters 7 and 8)
- Formatting printed reports and example user-defined functions used for formatting reports (Chapter 9)
- Creating and using data files for your business applications (Chapter 10)
- Determining what to do when your program doesn't work (Chapter 11)

This manual is not intended to give you a complete description of the syntax and rules required for each APL command, function, or variable; you must use the *5110 APL Reference Manual* for this information.

This manual does not need to be read chapter by chapter. Instead, you can read the appropriate chapters as required. For example, you might read Chapter 9, *Printer Control* when you need to format a report.

This manual follows the convention that *he* means *he or she*.

Prerequisite Publication

IBM 5110 APL Introduction, SA21-9301

Related Publications

IBM 5110 APL Reference Manual, SA21-9303

IBM 5110 APL Reference Card, GX21-9304

IBM 5110 Customer Support Functions Reference Manual, SA21-9311

Chapter 1. Data Processing Concepts

INTRODUCTION

The IBM 5110 Models 1 and 2 are multipurpose data processing systems with a personal touch. Because of their compact size and large-scale capabilities, these systems provide solutions to problems for a wide variety of users. As a 5110 user, only you can determine the problems to be solved with your system. To help you make these determinations, this chapter contains general concepts of data processing. You may or may not choose to read this chapter, depending on your data processing knowledge and experience.

CONCEPTS

What can you expect a computer to do with information? How do you get information into a computer? How does a computer know what to do with your information? What final results can you expect from a computer? This section gives general answers to these questions.

Today the computer is doing many jobs, from accounting to predicting election results to guiding spaceships. It is often looked upon as some kind of magical machine, but the computer performs no magic. Everything a computer does is dependent on the people who use it and the instructions they supply. For every job you want a computer to do, you must give a step-by-step procedure (a program) for it to follow. This procedure is then stored inside the computer. The information you want is processed according to the stored instructions.

A computer can do only a few rather simple things, but it does them extremely well. It can retrieve, almost instantly, any item of information stored in it. It can compare any two items of information, and do any arithmetic operations you want: add, subtract, multiply, or divide. It can be instructed to do any combination of these things in any sequence you want them done.

The computer works methodically, doing one thing at a time. When it finishes one step, it goes on to the next, then the next, and the next, according to instructions. But it performs these steps at an almost unbelievable speed until it comes up with the answer you want.

The task performed by a computer is called data processing. Data processing means that information is handled according to a set of rules. Whether you process information by hand or use a computer, the requirements of a job remain about the same. You must have *input*, which is the data you want to do something with; you must *process* the data, which is the act of doing something with data according to instructions; and you must have *output*, which is the result of your processing.

To help you understand the 5110 and data processing, let's first look at how a clerk might process information for the job of billing. For this job, assume the clerk works with the following data:

- Customer orders
- Price catalogs
- Customer records
- Accounts receivable records
- Inventory files

The clerk receives a copy of the customer orders after orders are shipped. He uses these documents to prepare bills that he sends to customers. To prepare the bill, he follows this procedure:

1. Look up, in a price catalog, the price of each item on the order.
2. Multiply the price by the quantity shipped.
3. Add the total price of items to get the total amount of bill.
4. Check the customer records to see if any special discounts apply, and adjust the bills accordingly.
5. Type the bill.
6. Adjust the accounts-receivable records to show what the customer owes.
7. Update the inventory records to show the reduced stock.

For each billing, the clerk follows the same procedure. In computer terms, this procedure is his *program* for doing the job. The customer order is his *input*, the calculating and file updating he does is *processing*, and the results of the processing (the billing and updated records) are his *output*.

The 5110 can speed up the billing operation and reduce costly errors. The order information can be entered from the keyboard; the records (such as price lists, customer records, accounts receivable records, and inventory files) can be quickly referenced and updated (processed) using tape or diskette storage; and the printer can print the billings.

The parts of the 5110 used for data processing are:

Input

- The keyboard from which data is entered into the system.
- Tape or diskette storage from which data can be read for processing.

Processing

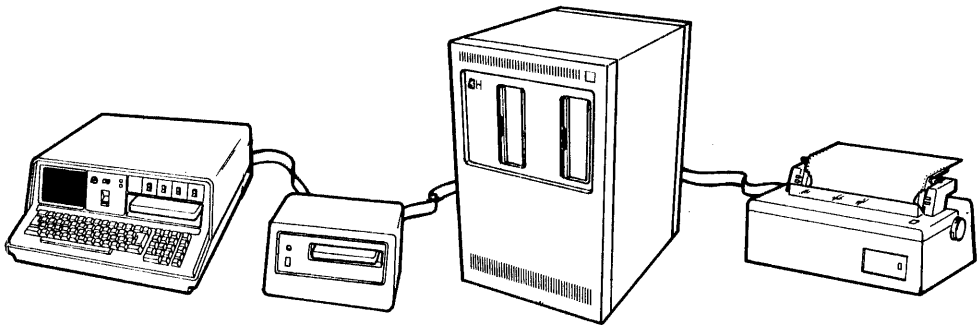
- The 5110 internal storage, which includes the active workspace. The active workspace is where calculations are performed and where user-defined functions (programs) and variables are stored.

Output

- The display screen, which displays keyboard input and the results of executed expressions or statements. That is, the display screen is a means of communication between the system and you.

- The printer, which prints keyboard input (if specified) and the results of executed expressions or statements. This printed output is sometimes called *hard copy* output.

- The tape or diskette on which data can be stored for future processing.



As mentioned before, a program is a procedure or set of instructions you establish for doing a job. These instructions are necessary because a computer cannot think for itself. When defining a program for the 5110, you can use a programming language called APL. APL is a simple-to-use programming language with which you describe how you want the 5110 to do a job. Also, in APL, a program is called a *user-defined* function.

The next chapter presents an approach to analyzing a problem or job so that an APL user-defined function can be used to help process information.

Chapter 2. Information Processing

This chapter presents an approach to dissecting an application so that APL user-defined functions can be used to help process the information. This approach helps you break down the application into manageable parts so that you can apply APL expressions and commands to process the information. Breaking the application down into manageable parts promotes thoroughness and allows the application to be solved (programmed) faster.

INPUT, PROCESS, AND OUTPUT

Every application consists of three parts:

- The *input* required for processing.
- The *process* (APL expressions) required to generate the final result
- The *output*, which is the final result

Each part might consist of one or more APL expressions. In the following sections, each part is discussed in more detail. Also, an example for finding the *compound interest* is used to illustrate each part.

Output

Because the output or result is the primary reason for a user-defined function to exist, considering the output provides the best place to start solving a problem. To do this, consider these questions:

1. What results are required?

2. How should the results be formatted?

3. Who uses the results? For example, should the results be displayed or printed, or should the results be stored in the active workspace, on tape, or on diskette for later use?

Now, for the compound interest example, assume the answers to these questions are:

1. The amount of interest earned.

2. The message THE INTEREST EARNED IS: followed by the calculated interest earned.

3. Finance officers will use the displayed results to evaluate different plans.

Once you have answered these questions, you know the purpose of a user-defined function.

Input

After the output, you should consider what input data is required to generate the output. To do this, consider such questions as:

1. What input is required?

2. Where does the input come from?

3. How is the input provided?

Continuing with the compound interest example, the answers to these questions are:

1. The interest rate, number of years, and principal.

2. From finance officers who need to know the amount of interest earned for different plans.

3. Through the 5110 keyboard.

In our example, most of the input data will come from the keyboard; however, other ways also exist. For example, some data might be permanent and be included within the user-defined function (for example, headings and labels). There might also be data that is usually constant but, for certain problems, must be changed. This data might be coded in the user-defined functions as variables that can be modified. And, of course, data might also be from tape or diskette.

The following list summarizes the input and output considerations so far:

	Data	Device
Input	Interest rate Number of years Principal	Keyboard
Output	THE INTEREST EARNED IS: The calculated interest earned	Display Screen

Process

Your introduction to APL started with APL's processing ability. The APL language is designed to do processing with a minimum number of instructions.

Once the input and output are well defined, all of the characteristics work together to make the process part the most straightforward.

For our compound interest example, the process part consists of:

1. Defining the algorithm used to calculate the compound interest

2. Using the input to generate the results

The formula used in this example for the compound interest is:

$$CI = Principal \times (1 + .01 \times Interest\ Rate)^{years}$$

The APL statements that use the input to generate the results might be as follows:

```
A←1+.01×INTERESTRATE
B←A*YEARS
C←PRINCIPAL×B
```

PUTTING IT ALL TOGETHER

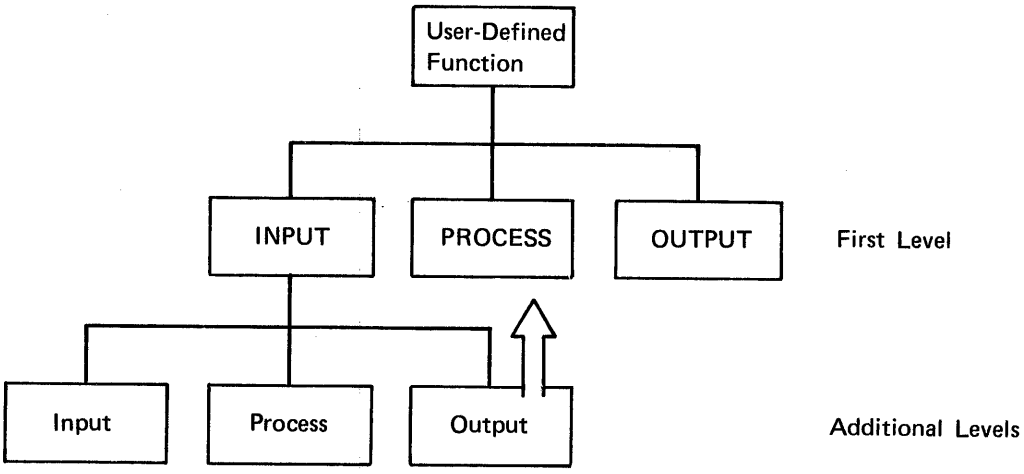
Now that you have considered the three parts of the application, it is time to write your user-defined function. For the compound interest example, your user-defined function might look like this:

```
     ∇ R←PRINCIPAL COMPOUND YEARS
[1]    A←1+0.01×INTERESTRATE
[2]    B←A*YEARS
[3]    CI←PRINCIPAL×B
[4]    R←'THE INTEREST EARNED IS: ', 10 2 ▼CI
     ∇
```

The interest rate must be assigned to this variable before the user-defined function is executed.

So far, you have taken a simple application and designed a user-defined function to solve it. If the application is larger or more complex, a more detailed structure is required. This more detailed structure involves expanding each of the three parts (input, process, and output) into additional levels of input, process, and output.

ADDITIONAL LEVELS OF INPUT, PROCESS, AND OUTPUT



The method shown in the previous diagram breaks the first-level INPUT, part into manageable parts. Let's continue with the compound interest problem and treat the INPUT portion as a separate problem in itself.

First, consider the output of the INPUT portion. Here the output is actually the input for the first-level PROCESS portion. In this case, assume that the output must be an interest rate not greater than 18%, a number of years not greater than 40, and a principal not greater than 500000.00.

Next, consider the input for the INPUT portion. The input is the same as before (the interest rate, number of years, and principal for which the interest earned must be calculated). However, in this case, the finance officers might be unfamiliar with the user-defined function; therefore, there should be prompting messages telling them what to enter.

Finally, consider the process for the INPUT portion. In this case, the processing consists of error checking and validation of all the input data, because you want to make sure that the interest rate is not greater than 18%, the number of years is not greater than 40, and the principal is not greater than 500000.00

Now, taking these considerations into account, the APL statements for the first-level input portion might be:

```
     ∇ R←EXAMPLE
[1]   START:'ENTER THE INTEREST RATE, YEARS, AND PRINCIPAL '
[2]     X←⍞
[3]     →((X[1]>18),(X[2]>40),(X[3]>500000))/E1,E2,E3
[4]     →PROCESS
[5]   E1:'THE INTEREST RATE IS GREATER THAN 18 PERCENT'
[6]     →START
[7]   E2:'THE NUMBER OF YEARS IS GREATER THAN 40'
[8]     →START
[9]   E3:'THE PRINCIPAL IS GREATER THAN 500000.00'
[10]    →START
[11]  PROCESS:
     ∇
```

As you break an application down into more manageable parts, you might want to have a separate user-defined function for each part. For example:

```
     ∇ R←INTEREST
[1]    R←INPUT
[2]    R←PROCESS R
[3]    R←OUTPUT R
     ∇


     ∇ X←INPUT
[1]   START:'ENTER THE INTEREST RATE, YEARS, AND PRINCIPAL '
[2]    X←⍬
[3]    →((X[1]>18),(X[2]>40),(X[3]>500000))/E1,E2,E3
[4]    →0
[5]   E1:'THE INTEREST RATE IS GREATER THAN 18 PERCENT'
[6]    →START
[7]   E2:'THE NUMBER OF YEARS IS GREATER THAN 40'
[8]    →START
[9]   E3:'THE PRINCIPAL IS GREATER THAN 500000.00'
[10]   →START
     ∇


     ∇ CI←PROCESS INPUT;A;B
[1]    A←1+0.01×INPUT[1]
[2]    B←A*INPUT[2]
[3]    CI←INPUT[3]×B
     ∇


     ∇ R←OUTPUT CI
[1]    R←'THE INTEREST EARNED IS: ', 10 2 ⍕CI
     ∇
```

CONCLUSION

When programming for an application using the 5110, break the application down into manageable parts. To do this, first focus on the program output. This is the primary interface to the user. It also defines what the real purpose of the program is. Next, consider all the input data that is required to generate the output. Finally (and only then) plan the actual processing.

Thinking in this way should help you make the transition from knowing the APL language to being able to use the APL language to generate user-defined functions for specific applications.

Chapter 3. APL Language Elements

In this chapter, the following topics concerning the APL language elements are discussed:

- Variables
- Data representation
- Arrays
- Examples of useful APL statements and user-defined functions

VARIABLES

You can store data in the 5110 by assigning it to a variable name. These stored items are called variables. Whenever the variable name is used, APL supplies the data associated with that name. A variable name can be up to 77 characters in length with no blanks; the first character must be alphabetic, and the remaining characters can be any combination of alphabetic, and numeric characters. The ← (assignment arrow) is used to assign data to a variable:

```
LENGTH←6
WIDTH←8
AREA←LENGTH×WIDTH
```

To display the value of a variable, enter the variable name:

```
      LENGTH
6
      WIDTH
8
      AREA
48
```

DATA REPRESENTATION

Numbers

The decimal digits 0 through 9 and the decimal point are used in the usual way. The character ¯, called the negative sign, is used to denote negative numbers. It appears as the leftmost character in the representation of any number whose value is less than zero:

```
      0 ¯4
¯4
      ¯3 ¯2
¯1
```

The negative sign, ¯, is distinct from - (the symbol used to denote subtraction) and can be used only as part of the numeric constant.

Scaled Representation (Scientific Notation)

You can represent numbers by stating a value in some convenient range, then multiplying it by the appropriate power of 10. This type of notation is called scaled representation in APL. The form of a scaled number is a number (multiplier) followed by E and then an integer (the scale) representing the appropriate power of 10. For example:

Number	Scaled Form
	↓——— Multiplier
66700	6.67E4
	↑——— Scale
.00284	2.84E¯3

The E (E can be read *times 10 to the*) in the middle indicates that this is scaled form; the digits to the right of the E indicate the number of places that the decimal point must be shifted. There can be no spaces between the E and the numbers on either side of it.

Numeric Value Range

Numeric values in the 5110 can range from ¯7.237005577332262E75 to 7.237005577332262E75. The smallest numeric value the 5110 can use is ±5.397604346934028E ¯79.

Numeric Value Precision

Numbers in the 5110 are carried internally with a precision of 16 significant digits.

Character Constants

Zero or more characters enclosed in single quotes, including overstruck characters and blank characters (spaces), is a character constant. The quotes indicate that the characters keyed do not represent numbers, variable names, or functions, but represent only themselves. When character constants are displayed, the enclosing quotes are not shown:

```
      'ABCDEFG'
ABCDEFG
      '123ABC'
123ABC
      M←'THE ANSWER IS:'
      M
THE ANSWER IS:
```

When a quote is required within the character constant, a pair of quotes must be entered to produce the single quote in the character constant. For example:

```
      'DON''T GIVE THE ANSWER AWAY'
DON'T GIVE THE ANSWER AWAY
```

Logical Data

Logical (Boolean) data consists of only ones and zeros. The relational function (> ≥ = < ≤ ≠) generate logical data as their result; the result is 1 if the condition was true and 0 if the condition was false. The output can then be used as arguments to the logical functions (∧ ∨ ⍲ ⍱) to check for certain conditions being true or false. Logical data can also be used with the arithmetic functions, in which case it is treated as numeric 1's and 0's.

ARRAYS

Array is the general term for a collection of data, and includes scalars (single data items), vectors (strings of data), matrices (tables of data), and arrays of higher dimensions (multiple tables). All primitive (built-in) functions are designed to handle arrays. Some functions are designed specifically to handle arrays rather than scalars. Indexing, for example, can select certain elements from an array for processing.

One of the simplest kinds of arrays, the vector, has only one dimension; it can be thought of as a collection of elements arranged along a horizontal line. The numbers that indicate the positions of elements in an array are called indices. An element can be selected from a vector by a single index because a vector has only one dimension. The following example shows assigning a numeric and a character vector to two variable names, N and C; the names are then entered to display the values they represent:

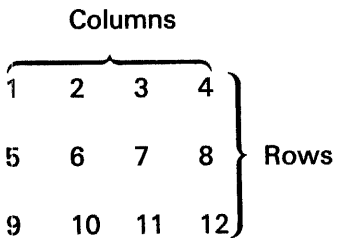
```
      N←5 6.2 ¯3 888 95.12
      N
5 6.2 ¯3 888 95.12
      C←'ABCDEFG'
      C
ABCDEFG
```

Generating Arrays

The most common way to generate an array is to specify the shape the array is to have (that is, the length of each coordinate) and the values of the elements of the new array. The APL function that forms an array is the reshape function. The symbol for the reshape function is ρ. The format of the function used to generate an array is XρY, where X is the shape of the array and Y represents the values for the elements of the array. For the left argument (X), you enter a number for each coordinate to be generated; this number indicates the length of the coordinate. Each number in the left argument must be separated by at least one blank. The values of the elements of the new array are whatever you enter as the right argument (Y). The instruction 7ρA means that the array to be generated has one dimension (is a vector) seven elements in length, and that seven values are to be supplied from whatever values are found stored under the name A. It does not matter how many elements A has, as long as it has at least one element. If A has fewer than seven elements, its elements are repeated as often as needed to provide seven entries in the new vector. If A has more than seven elements, the first seven are used. The following examples show generation of some vectors:

```
      7ρ1 2 3
1 2 3 1 2 3 1
      2ρ123
123 123
      5ρ1.3
1.3 1.3 1.3 1.3 1.3
```

An array with two coordinates (rows and columns) is called a matrix.



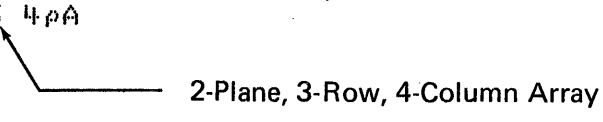
To generate a matrix, you specify X (left argument) as two numbers, which are the lengths of the two coordinates. The first number in X is the length of the first coordinate, or number of rows, and the second number is the length of the second coordinate, or number of columns. The following example shows how a matrix is generated:

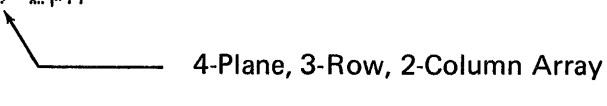
```
      M←2 3⍴1 2 3 4 5 6
      M
1 2 3
4 5 6
      M←2 4⍴'ABCDEFGH'
      M
ABCD
EFGH
      M1←2 3⍴M
      M1
ABC
DEF
```

Note that the values in the right argument are arranged in row order in the arrays. If the right argument has more than one row, the elements are taken from the right argument in row order.

The rank of an array is the number of coordinates it has, or the number of indices required to locate any element within that array. Scalars are rank 0. Vectors have a rank of 1, matrices have a rank of 2, and N-rank arrays have a rank from 3 to 63 (where N is equal to the rank). N-rank arrays, like matrices, are generated by providing as the left argument a number indicating the length for each coordinate (for example, planes, rows, and columns). The following examples show how to generate 3-rank arrays. Note that the elements taken from the right argument are arranged in row order:

```
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      2 3 4⍴A
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

      4 3 2⍴A
AB
CD
EF

GH
IJ
KL

MN
OP
QR

ST
UV
WX
```





Finding the Shape and Rank of An Array

Once you have generated an array, you can find its shape (number of elements in each coordinate) by specifying ρ (shape function) with only a right argument, which is the name of the array. If A is a vector with six elements and you enter ρA, the result is one number because A is a one-dimensional array. The number is 6, the length (number of elements) of A. The result of the shape function is always a vector:

```
      A←111 222 333 444 555 666
      ρA
6
```

The shape of a matrix or N-rank array is found the same way:

```
      M←2 3ρ1 2 3 4 5 6
      M
1 2 3
4 5 6
      ρM
2 3
      R←2 3 4ρ1 2 3 4 5 6 7 8
      R
1 2 3 4
5 6 7 8

1 2 3 4
5 6 7 8
      ρR
2 3 4
```

In some cases, it might be necessary to know just the rank, the number or coordinates (or indices) of an array. To find the rank, enter ρρ (shape of the shape) and a right argument, which is the name of the array:

```
      A←111 222 333 444 555 666
      B←2 3ρ1 2 3 4 5 6
      C←2 3 4ρ1 2 3 4 5 6 7
      ρA
6
      ρρA
1
      ρB
2 3
      ρρB
2
      ρC
2 3 4
      ρρC
3
```

The following table shows what the shapes and ranks are for the various types of arrays:

Data Type	Shape ρX	Rank ρρX
Scalar	No dimension (indicated by an empty vector).	0
Vector	Number of elements.	1
Matrix	Number of rows and the number of columns.	2
N-rank arrays	Each number is the length of a coordinate.	N

Empty Arrays

Although most arrays have one or more elements, arrays with no elements also exist. An array with no elements is called an *empty array*. Empty arrays are useful when you are creating lists (see *Catenation* in this chapter) or branching in a user-defined function.

Following are some ways to generate empty arrays:

- Assign ⍳0 to a variable name to generate an empty vector:

```
      EVECTOR←⍳0
      EVECTOR
←──────────────────────  An empty array is indicated
                         by a blank display.
      ρEVECTOR
0
↖──────────────────────  The shape of the empty vector
                         is zero (zero elements).
```

- Use a zero length coordinate when generating a multidimensional array:

```
      EMATRIX1←3 0ρ⍳0 ←──  This matrix has three rows and no (0)
      EMATRIX1             columns. If one of the coordinates is
                           not zero, you cannot generate the
                           empty array.
←──────────────────────
      ρEMATRIX1
3 0                    ↖──  A Blank Output Display
```

- A function might generate an empty vector as its result; for example, finding the shape of a scalar:

```
      ρ'A'
←──────────────────────  A Blank Output display.
```

INDEXING ARRAYS

You may not want to refer to the whole array but just to certain elements. Referring to only certain elements is called indexing. Index numbers must be integers; they are enclosed in brackets and written after the name of the variable to which they apply. Assume that A is assigned a vector as follows: A ← 11 12 13 14 15 16 17. The result of entering A is the whole vector, and the result of entering A[2] is 12 (assuming the index origin is 1).

Here are some more examples of indexing:

```
      A←11 12 13 14 15 16 17
      A[3]
13
      A[5 3 7 1]
15 13 17 11
      B←3 1 4 6
      A[B]
13 11 14 16
      B←'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
      B[4 1 14 27 1 14 4 27 3 12 1 9 18]
DAN AND CLAIR
      C←22 9 18 7 9 14 9 1
      B[C]
VIRGINIA
```

Blank Character

If you use an index that refers to an element that does not exist in the array, the instruction cannot be executed and INDEX ERROR results:

```
      A
11 12 13 14 15 16 17
      A[8]
INDEX ERROR
      A[8]
      ^
```

You cannot index or do anything else with an array until after the array has been specified. For example, suppose that no value has been assigned to the name Z; then an attempt to store values in certain elements within Z would result in an error, because those elements do not exist:

```
      Z[3 4]←18 46
VALUE ERROR
      Z[3 4]← 18 46
      ^
```

Indices (whatever is inside the brackets) can be expressions, provided that when those expressions are finally evaluated, the results are values that represent valid indices for the array:

```
      B
ABCDEFGHIJKLMNOPQRSTUVWXYZ
      X←1 2 3 4 5
      B[X×2]
BDFHJ
      X
1 2 3 4 5
      B[1+X×3]
DGJMP
```

The array from which elements are selected does not have to be a variable. For example, a vector can be indexed as follows:

```
      2 3 5 7 9 11 13 15 17 19[7 2 4 2]
13 3 7 3
      'ABCDEFGHIJKLMNOPQRSTUVWXYZ '[12 15 15 11 27 16 1]
LOOK PA
      'ABCDEFGHIJKLMNOPQRSTUVWXYZ '[2 4⍴4 15 14 27 13 1 18 25]
DON
MARY
```

Indexing a matrix or N-rank array requires an index number for each coordinate. The index numbers for each coordinate are separated by semicolons. Suppose M is a 3 by 4 matrix of consecutive integers:

```
M←3 4⍴⍳12
```

If you ask to see the values of M, they are displayed in the usual matrix form:

```
      M
1  2  3  4
5  6  7  8
9 10 11 12
```

If you want to refer to the element in row 2, column 3, you enter:

```
      M[2;3]
7
```

If you want to refer to the third and fourth elements in that row, you enter:

```
      M[2;3 4]
7 8
```

Similarly, to refer to the elements in column 4, rows 1, 2, and 1, you enter:

```
      M[1 2 1;4]
4 8 4
```

You can use the same procedure to select a matrix within a matrix. If you want the matrix of those elements in rows 2 and 3 and columns 1, 2, and 1 of M, you enter:

```
    M[2 3;1 2 1]
5  6  5
9 10  9
```

If you do not specify the index number for one or more of the coordinates of the array that you are indexing, APL assumes that you want the entire coordinate(s). For instance, to get all of row 2, you enter:

```
      M[2;]
5 6 7 8
```

Or to get all of columns 4 and 1, you enter:

```
     M[;4 1]
 4  1
 8  5
12  9
```

Note: You still have to enter the semicolon to make clear which coordinate is which. The number of semicolons required is the rank of the array minus one. If the correct number of semicolons is not specified, RANK ERROR results:

```
      M←3 4⍴⍳12
      ⍴M
3 4
      M[6]←9
RANK ERROR
      M[6]←9
      ^
```

You can change elements within an array by assigning new values for the indexed elements. (The rest of the array remains unchanged.)

```
      A←3 3⍴1 2 3 4 5 6 7 8 9
      A
1 2 3
4 5 6
7 8 9
      A[2;2 3]←10 20
      A
 1  2  3
 4 10 20
 7  8  9
```

JOINING ARRAYS TOGETHER

You can join two arrays to make a single array by using the catenation or lamination functions. The symbol for these functions is the comma.

Catenation

When catenating vectors, or scalars and vectors, the variables are joined in the order in which they are specified. For example:

```
      A←1 2 3 4
      B←4 5 6
      A,B
1 2 3 4 4 5 6
      B,A
4 5 6 1 2 3 4
      A,2
1 2 3 4 2
      3,A
3 1 2 3 4
```

When catenating two matrices or N-rank arrays, the function can take the form A,[I]B, where I defines the coordinate that will be expanded when A and B are joined. If the coordinate is not specified, the last coordinate is used. When A and B are matrices and [I] is [1], the first coordinate (number of rows) is expanded; when [I] is [2], the last coordinate (number of columns) is expanded. The following examples show how to catenate matrices:

Graphic Representation

```
      A←2 3⍴10 20 30 40 50 60
      B←2 3⍴11 22 33 44 55 66
```

A

10	20	30
40	50	60

B

11	22	33
44	55	66

```
      A,B
10 20 30 11 22 33
40 50 60 44 55 66
```

A

10	20	30	11	22	33
40	50	60	44	55	66

B

```
      A,[2]B
10 20 30 11 22 33
40 50 60 44 55 66
```

```
      A,[1]B
10 20 30
40 50 60
11 22 33
44 55 66
```

A

10	20	30
40	50	60

B

11	22	33
44	55	66

Arrays of unequal sizes can be catenated, provided that the lengths of the coordinates not specified are the same (see the first example following). If the coordinates not specified have different values, an error results (see the second example following):

```
      A←2 3⍴10 20 30 40 50 60
      B←2 4⍴11 22 33 44 55 66 77 88
```

10	20	30
40	50	60

11	22	33	44
55	66	77	88

```
      A,[2]B
10 20 30 11 22 33 44
40 50 60 55 66 77 88
```

A

10	20	30	11	22	33	44
40	50	60	55	66	77	88

B

```
      A,[1]B
LENGTH ERROR
      A,[1] B
      ^
```

A

10	20	30	
40	50	60	

B

11	22	33	44
55	66	77	88

A scalar can also be catenated to an array. In the following example, a scalar is catenated to a matrix. Notice that the scalar is repeated to complete the coordinate:

```
      A←2 3⍴10 20 30 40 50 60
      A
10 20 30
40 50 60
      A,[2]99
10 20 30 99
40 50 60 99
      A,[1]99
10 20 30
40 50 60
99 99 99
```

A vector can also be catenated to another array, provided the length of the vector matches the length of the coordinate *not* specified. See the following examples:

```
      A,99 88
10 20 30 99
40 50 60 88
```

10	20	30	99
40	50	60	88

```
      A,[1]99 88
LENGTH ERROR
      A,[1] 99 88
      ^
```

10	20	30
40	50	60
99	88	

The catenate function is useful when you are creating lists of information. Sometimes it is necessary to use an empty array to start a list. For example, suppose you want to create a matrix named PHONE where each row will represent a seven-digit telephone number. First you want to establish the matrix, then add the telephone numbers at a later time. The following instruction will establish an empty array named PHONE with no (0) rows and seven columns:

```
      PHONE←0 7⍴⍳0
      PHONE
      ρPHONE
0 7
```

Blank display indicates an empty array.

Now, the telephone numbers can be added as follows:

```
      PHONE←PHONE,[1]'5336686'
      PHONE
5336686
      PHONE←PHONE,[1]'4564771'
      PHONE
5336686
4564771
      ρPHONE
2 7
```

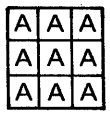
The list of telephone numbers now contains two rows.

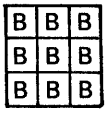
Lamination

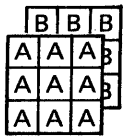
When laminating two variables together, the function joins the variables together by creating a *new* coordinate. The function takes the form A,[I]B, where I is an index number that must be a fraction. This index number specifies where the new coordinate is added. If the index number is less than 1, the new coordinate is added before the first coordinate; if the index number is between 1 and 2, the new coordinate is added between the first and second coordinate; and so on. For example:

Graphic Representation

```
      A←3 3⍴'A'
      A
AAA
AAA
AAA
      B←3 3⍴'B'
      B
BBB
BBB
BBB

      C←A,[.1]B
      C
AAA
AAA
AAA

BBB
BBB
BBB
      ⍴C
2 3 3
```

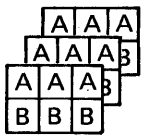






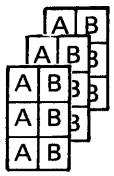
The new coordinate is added before the first coordinate.

```
      C←A,[1.1]B
      C
AAA
BBB

AAA
BBB

AAA
BBB

      ⍴C
3 2 3
```



The new coordinate is added between the first and second coordinate.

```
      C←A,[2.1]B
      C
AB
AB
AB

AB
AB
AB

AB
AB
AB

      ⍴C
3 3 2
```

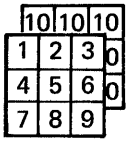


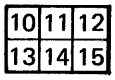
The new coordinate is added after the second coordinate.

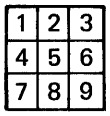
The new coordinate is always 2 because *two* variables are joined along a new coordinate.

Unless one of the variables is a scalar, arrays of unequal sizes cannot be laminated together. For example:

```
      A←3 3⍴⍳9
      A
1 2 3
4 5 6
7 8 9
      C←A,[.1]10
      C
 1  2  3
 4  5  6
 7  8  9

10 10 10
10 10 10
10 10 10
      B←2 3⍴10 11 12 13 14 15
      B
10 11 12
13 14 15
      B,[.1]A
LENGTH ERROR
      B,[0.1] A
      ^
```







USEFUL APL STATEMENTS AND USER-DEFINED FUNCTIONS

To remove duplicate blanks from a character vector:

```
      ∇COMPRESS[⎕]∇
    ∇ Z←COMPRESS W;I
[1]   Z←1↓((1⌽I)⌿I←W≠' ')/W←' ',W
    ∇

      COMPRESS 'AAA              BBBB     CCC    DDD'
AAA BBBB CCC DDD
```

The right argument is the character vector.

To create a matrix from a character vector with a delimiter for each row:

```
      ∇FORM[⎕]∇
    ∇ M←D FORM S;A;B;X;Z;⎕IO
[1]   ⎕IO←0
[2]   M←(X←S≠D)/S←,S
[3]   Z←(X≠1↓X,~¯1↑X)/1+⍳⍴X
[4]   M←((⍴A),B)⍴(,(A-1)∘.≥⍳B←0⌈⌈/A←X[Z-1]/Z-0,¯1↓Z)\M
    ∇

      '∘' FORM 'A∘B∘CCC∘D'
A
B
CCC
D
```

⎕IO will be set to 0 just during the execution of this function.

In this example, the ∘ character is used as the delimiter. The left argument specifies the delimiter, and the right argument specifies the character vector.

To remove the alphabetic characters from a character vector, leaving only the numeric characters and blanks:

```
      ∇REMOVE[⎕]∇
    ∇ NEW←REMOVE OLD
[1]   NEW←(OLD∈'0123456789. ')/OLD
    ∇

      OLD←'DAN 30 DAVE 29.5 JERRY 37 MEL 40.8'
      REMOVE OLD
 30  29.5  37  40.8
```

To replace all trailing blanks with a minus sign:

```
      ∇BLANKS[⎕]∇
    ∇ Z←BLANKS M;V
[1]   V←,Φ∧\ΦM=' '
[2]   Z←,M
[3]   Z[V/⍳ρV]←'-'
[4]   Z←(ρM)ρZ
    ∇

      MAT←2 5ρ'AB   CDE  '
      MAT
AB
CDE
      BLANKS MAT
AB---
CDE--
```

Return all elements of a vector that are even numbers:

```
      ∇EVEN[⎕]∇
    ∇ Z←EVEN V
[1]   Z←(0=2|V)/V
    ∇

      EVEN 0 1 2 3 4 5 6 7 8 9
0 2 4 6 8
```

To delete all comment lines from a user-defined function:

```
      ∇COMMENT[⎕]∇
    ∇ COMMENT FN;M
[1]   M←⎕FX(M[;1]≠'⍝')/M←⎕CR FN
    ∇

      ∇ADD
[1] ⍝COMMENT LINE 1
[2] 3+4
[3] ⍝COMMENT LINE 2
[4] ∇

      COMMENT 'ADD'
      ∇ADD[⎕]∇
    ∇ ADD
[1]   3+4
    ∇
```

To merge two variables with the same shape into a single vector:

```
      ∇MERGE[⎕]∇
     ∇ Z←A MERGE B
[1]    Z←,A,[⎕IO-0.5] B
     ∇

      A←2 2⍴'A'
      B←2 2⍴'B'

      A MERGE B
AAAABBBB
```

To delete duplicate elements in a vector:

```
      ∇DUPLICATE[⎕]∇
     ∇ Z←DUPLICATE V
[1]    Z←((⍳⍴V)=V⍳V)/V←,V
     ∇

      DUPLICATE 1 2 3 3 4 2 5 6 5 5
1 2 3 4 5 6
```

To find the first nonblank character in a character vector:

```
      ∇NONBLANK[⎕]∇
     ∇ Z←NONBLANK W
[1]    Z←1↑(W≠' ')/W
     ∇

      V←'        DAN'

      NONBLANK V
D

      ∇NONBLANKX[⎕]∇
     ∇ Z←NONBLANKX W
[1]    Z←(<\W≠' ')/W
     ∇

      NONBLANKX V
D
```

To determine whether a variable is character or numeric:

```
      ∇DETERMINE[⎕]∇
    ∇ DETERMINE W
[1]   'NC'[⎕IO+(¯1↑2↑1↑,W)=' ']
    ∇

      A←'9'
      B←9

      DETERMINE A
C
      DETERMINE B
N

      ∇DETERMINE1[⎕]∇
    ∇ DETERMINE1 W
[1]   'NC'[⎕IO+' '=1↑0⍴W]
    ∇

      DETERMINE1 A
C
      DETERMINE1 B
N
```

The result is C for character or N for numeric.

To make scalar or vector into a matrix:

```
      ∇MATRIX[⎕]∇
    ∇ Z←MATRIX M
[1]   Z←(¯2↑1,1,⍴M)⍴M
    ∇

      A←'ABCDEFG'
      X← MATRIX A
      X
ABCDEFG
      ⍴X
1 7
```

The result is a 1 by 7 matrix.

To delete all user-defined function names starting with a specified character vector from the active workspace:

```
      ∇DELETEFN[⎕]∇
    ∇ DELETEFN C;NL;Z;X
[1]   Z←((((1↑⍴NL),⍴C)↑NL)∧.=C←,C)/NL←(1↑C) ⎕NL 3
[2]   X←⎕EX Z
    ∇

      ∇ADD[⎕]∇
    ∇ ADD
[1]   3+4
    ∇

      DELETEFN 'AD'
      ADD
VALUE ERROR
      ADD
      ^
```

A user-defined function in the active workspace

The function is no longer in the active workspace

To count the number of occurrences of each unique character in a character vector:

```
      ∇COUNT[⎕]∇
     ∇ Z←COUNT W;UC
[1]    Z←+⌿W∘.=(⎕AV∈W)⌿⎕AV
     ∇

      COUNT 'ABBCCCDDDDE'
1 2 3 4 1
      COUNT 'ABCDBCDCDD'
1 2 3 4
```

To center the character string in each row of a character matrix:

```
      ∇CENTER[⎕]∇
     ∇ Z←CENTER M
[1]    Z←(-⌈(+/∧\ΦM=' ')÷2)ΦM←(+/∧\M=' ')ΦM
     ∇

      MAT←3 6⍴' A     B     C'
      MAT
 A
B
     C

      CENTER MAT
   A
   B
   C
```

To right-justify the character string in each row of a character matrix:

```
      ∇RIGHTJUSTIFY[⎕]∇
     ∇ Z←RIGHTJUSTIFY M
[1]    Z←(-+/∧\(ΦM)=' ')ΦM
     ∇

      MAT
 A
B
     C

      RIGHTJUSTIFY MAT
     A
     B
     C
```

To left-justify the character string in each row of a character matrix:

```
      ∇LEFTJUSTIFY[⎕]∇
    ∇ Z←LEFTJUSTIFY M
[1]   Z←(+/∧\M=' ')⌽M
    ∇

      MAT
 A
B
    C

      LEFTJUSTIFY MAT
A
B
C
```

To list each user-defined function in the active workspace:

```
      ∇LISTFNS[⎕]∇
    ∇ LISTFNS;ALF;NAME;I;FUN;COL;NO;⎕IO;⎕PW
[1]   ⍝THIS FUNCTION LISTS ALL FUNCTIONS IN THE ACTIVE WORKSPACE
[2]   ⍝EXCEPT LISTFNS AND LISTVARS.
[3]    ⎕IO←1
[4]    ⎕PW←132
[5]    NAME←⎕NL 3
[6]    ALF←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[7]    ALF←ALF,'ΔABCDEFGHIJKLMNOPQRSTUVWXYZΔ'
[8]    ALF←ALF,'0123456789'
[9]    NAME←NAME[⍋66↓ALF⍳⍷NAME;]
[10]   I←0
[11] LOOP:→((1↑⍴NAME)<I←I+1)/0
[12]   →(∧/NAME[I;]=(¯1↑⍴NAME)↑'LISTFNS')/LOOP
[13]   →(∧/NAME[I;]=(¯1↑⍴NAME)↑'LISTVARS')/LOOP
[14]   NO←1↑⍴FUN←⎕CR NAME[I;]
[15]   COL←((NO,1)⍴'['),(⍕(NO,1)⍴¯1+⍳NO),((NO,2)⍴'] ')
[16]   COL[1;]←(¯1↑⍴COL)↑' '
[17]   COL,FUN
[18]   2 1 ⍴' '
[19]   →LOOP
    ∇

      LISTFNS
     Z←BLANKS M;V
[ 1] V←,⌽∧\⌽M=' '
[ 2] Z←,M
[ 3] Z[V/⍳⍴V]←'-'
[ 4] Z←(⍴M)⍴Z


     Z←A BY B;ROW
[ 1] A←(2↑(⍴A), 1 1)⍴A

LISTFNS[18] ←
```

The ATTN key was pressed before all the functions in the active workspace were listed.

To list each variable and associated value in the active workspace:

```
      ∇LISTVARS[⎕]∇
    ∇ LISTVARS;ALF;VAR;I;R;⎕IO
[1]   ⍝THIS FUNCTION LIST THE VARAIBLES IN THE ACTIVE WORKSPACE.
[2]    ⎕IO←1
[3]    R←1↑⍴VAR←⎕NL 2
[4]    ALF←' ABCDEFGHIJKLMNOPQRSTUVWXYZΔ'
[5]    ALF←ALF,'ABCDEFGHIJKLMNOPQRSTUVWXYZΔ'
[6]    ALF←ALF,'0123456789'
[7]    VAR←VAR[⍋66⍳ALF⍳⍉VAR;]
[8]    I←0
[9]   LOOP:→(R<I←I+1)/0
[10]   2 1 ⍴' '
[11]   ⍞←'       ',VAR[I;]
[12]   →LOOP
    ∇

      LISTVARS


      A
 10  20  30
 40  50  60


      B
 11  22  33
 44  55  66


      C
 AAA
  B

LISTVARS[12]
```

The ATTN key was pressed before all the variables in the active workspace were listed.

To field-protect high-order digits:

```
      ∇PROT[⎕]∇
    ∇ Z←N PROT V
[1]   Z←(-N)↑((N-3)⍴'*'),V
    ∇

      10 PROT '123.45'
****123.45
```

To delete leading blanks from a character string:

```
      ∇DLB[⎕]∇
    ∇ Z←DLB A
[1]   Z←(¯1+(A=' ')\0)↓A
    ∇

      DLB '          123.45'
123.45
```

To join vectors and print results as a single, sorted vector:

```
      ∇JOIN[⎕]∇
    ∇ Z←A JOIN B;T
[1]   Z←T[⍋T←A,B]
    ∇

      A←10 5 6 2 3 1
      B←15 5 2 2 7

      A JOIN B
1 2 2 2 3 5 5 6 7 10 15

      ∇JOIN1[⎕]∇
    ∇ Z←A JOIN1 B;T
[1]   Z←T[⍒T←A,B]
    ∇

      A JOIN1 B
15 10 7 6 5 5 3 2 2 2 1
```

To sort up to the first nine columns of a matrix with each row representing a name:

```
      ∇SORT[⎕]∇
    ∇ Z←SORT LIST;ALF
[1]   ALF←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[2]   Z←LIST[⍋29↑ALF⍳⍤1⊢LIST;]
    ∇

      L←6 7⍴'JENNY   JAN     LUCY    ARCHIE DOUGLASBETH
      L
JENNY
JAN
LUCY
ARCHIE
DOUGLAS
BETH
      L←SORT L
      L
ARCHIE
BETH
DOUGLAS
JAN
JENNY
LUCY
```

To find the location of a name in a list of names:

```
      ∇FIND[⎕]∇
    ∇ Z←LIST FIND NAME
[1]   Z←((LIST∧.=(⍉1↑⍴LIST)↑NAME))/⍳1↑⍴LIST
    ∇

      L FIND 'JAN'
4
```

To delete a name from a list of names:

```
      ∇DEL[⎕]∇
    ∇ Z←LIST DEL NAME
[1]   Z←((~LIST∧.=(⍉1↑⍴LIST)↑NAME))⌿LIST
    ∇

      L DEL 'JAN'
ARCHIE
BETH
DOUGLAS
JENNY
LUCY
```

To perform a specified operation if a condition is true:

```
        ∇IF[⎕]∇
     ∇  Z←OP IF COND
[1]     Z←COND/OP
     ∇

        I←3

        'PRINT' IF I=3
PRINT
        'PRINT' IF I=4
```

Chapter 4. Active Workspace Control

The active workspace is the internal storage where calculations are performed; it is also the place where variables and user-defined functions are stored. The 5110 system commands, system functions, and system variables are used to control the active workspace. In this chapter, the following topics are discussed:

- The active workspace *environment* in a CLEAR WS
- Getting information about the active workspace
- Changing the active workspace environment
- Getting data into the active workspace
- Active workspace storage considerations

THE ACTIVE WORKSPACE ENVIRONMENT IN A CLEAR WS

When the 5110 is first turned on, or the RESTART switch is pressed, or the)CLEAR command is executed, the active workspace environment has the following characteristics:

- The index origin (⎕IO) is 1.
- The comparison tolerance (⎕CT) is 1E-13.
- The random number seed (⎕RL) is 16807.
- The print width (⎕PW) is 64.
- The print precision (⎕PP) is 5.
- The latent expression (⎕LX) is an empty vector.
- The workspace identification [ )WSID] is CLEAR WS.
- The number of symbols allowed [ )SYMBOLS] is 125.

(See the *IBM 5110 APL Reference Manual*, SA21-9303, for a complete description of the system variables and system commands.)

These characteristics control the way some of the APL functions and system commands will work in the active workspace. For example, if you have assigned 125 variable names and you enter the statement:

```
      NAME126←'ROCHESTER'
SYMBOL TABLE FULL
           NAME126
                 ^
```

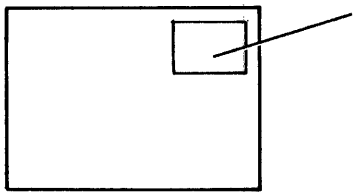
a SYMBOL TABLE FULL error message is displayed, because only 125 symbols (names) are initially allowed in the active workspace. How you change some of the active workspace environmental characteristics is discussed next.

CHANGING THE ACTIVE WORKSPACE ENVIRONMENT

You can change the system variables, workspace identification [ )WSID], and number of symbols allowed [ )SYMBOLS]. For the system variables, you simply assign them a new value. For example:

```
⎕IO←0 ◄———— The index origin is now 0
```

The number of symbols allowed in the active workspace can be established only in clear workspace. That is, the size of the *symbol table* must be established by the 5110 before any other data is placed in the active workspace.



Active Workspace

The number of symbols allowed is initially set to 125, which requires 1000 bytes of active workspace storage.

There might be times when you have used the maximum number of symbols allowed, so you may need to increase the maximum number of symbols allowed. If you change the number of symbols allowed in a clear workspace and then use the)LOAD command to load a stored workspace into the active workspace, the number of symbols allowed is the same as when the stored workspace was written to the media. However, you can use the following procedure to change the number of symbols allowed:

1. Save the contents of the active workspace using the)SAVE command.

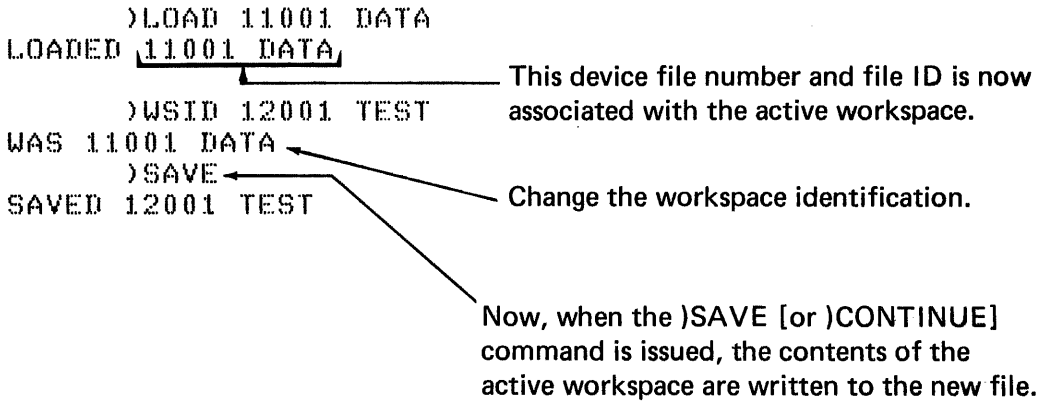
2. Clear the active workspace using the)CLEAR command.

3. Set the new number of symbols allowed using the)SYMBOLS command. For example:

```
      )SYMBOLS 251 ◄———— Now, 251 symbols are allowed.
WAS 125
```

4. Copy the stored workspace into the active workspace using the)COPY command. Using the)COPY command does not affect the number of symbols allowed in the active workspace. The)COPY command is discussed later in this chapter.

The workspace identification specifies the device/file number and file ID where the contents of the active workspace are stored when a)SAVE or)CONTINUE command is issued. The)WSID command can be used to change the device/file number and file ID where the contents of the active workspace is to be stored. For example:



GETTING DATA INTO THE ACTIVE WORKSPACE

You can get data into the active workspace by entering the data from the keyboard or reading the data from a tape or diskette file. You read data from a tape or diskette file using an APL shared variable or one of the following system commands:

-)LOAD
-)RESUME
-)COPY
-)PCOPY
-)PROC

See Chapter 10, *Input/Output Control*, for information on reading data using APL shared variables. The system commands used for reading data into the active workspace are discussed next.

The)LOAD and)RESUME Commands

The)LOAD and)RESUME commands are used to load an entire stored workspace into the active workspace. The contents of the stored workspace then replace the contents of the active workspace. The)LOAD command has the following characteristics:

- Loads any stored workspace, which was written to tape or diskette by the)SAVE or)CONTINUE command, into the active workspace.
- If the stored workspace has a latent expression assigned to the ⎕LX system variable, the latent expression is executed each time the)LOAD command is used to load that stored workspace into the active workspace.
- If the stored workspace has shared variables established, the shared variables are not reestablished when the)LOAD command loads the stored workspace into the active workspace.

The)RESUME command has the following characteristics:

- Loads any stored workspace which was written to tape or diskette by the)SAVE or)CONTINUE command into the active workspace.
- If the stored workspace has a latent expression, the latent expression is *not* executed when the continued (stored) workspace is loaded into the active workspace.
- The primary reason for using this command is to reestablish the system environment as it was when the workspace was written to the media. That is, if the stored workspace was written to the media by the)CONTINUE command, any shared variables and/or suspended functions in the stored workspace are reestablished in the active workspace by the)RESUME command. See *The)RESUME Command* in the *IBM 5110 APL Reference Manual*, SA21-9303, for a description of how shared variables are reestablished.

The)COPY and)PCOPY Commands

The)COPY and)PCOPY commands are used to copy all or selected objects (variables or user-defined functions) from a stored workspace into the active workspace. When these commands are used, the objects are copied into the active workspace without replacing the entire contents of the active workspace. When the)COPY command is used, any objects already in the active workspace are replaced by the objects specified by the)COPY command if the objects have the same name. When the)PCOPY command is used, any objects in the active workspace are *protected* and not replaced by the specified objects if the objects have the same name. These commands have the following characteristics:

- These commands can only be used to copy objects from a workspace written to tape or diskette by the)SAVE command. If you want to copy objects from a workspace written to tape or diskette by the)CONTINUE command, the stored workspace must be loaded into the active workspace and then written to the media by the)SAVE command.

- These commands cannot be used if the active workspace contains suspended functions.

- These commands can be used to copy objects from several stored workspaces into the active workspace.

The)PROC Command

An alternative to entering data from the keyboard is to get the data from a *procedure file*. A procedure file contains character records that represent any input that is possible from the keyboard, such as system commands, function definition, and APL expressions. When the)PROC command is issued, the 5110 reads and executes one procedure file record at a time until the last record (end-of-data) is processed. Then the 5110 goes back to using regular keyboard input. [See the *IBM 5110 APL Reference Manual*, SA21-9303, for a complete description of procedure files and the)PROC command.]

A procedure file must be a type I or U data file and the records cannot be greater than 128 characters. (See Chapter 10, *Input/Output Control*, for a complete description of data files.)

A procedure file is useful for doing unattended applications that require using system commands and/or function definition. For example, assume you have an application that requires several user-defined functions. However, not enough workspace storage is available to contain all of the user-defined functions. In this case, you might:

1. Use the)SAVE command to store the user-defined functions on tape or diskette.

2. Create a procedure file that contains the following character records:
   a. A)COPY command to copy the first user-defined function(s) required for the application into the active workspace
   b. The statement(s) required to execute the user-defined function(s)
   c. An)ERASE command that erases user-defined functions and variables that are no longer required
   d. A)COPY command that copies the next user-defined function(s) required for the application into the active workspace

   The previous steps are repeated until the application is complete.

3. Use the)PROC command to execute the statements from the procedure file. After the last statement is read and executed, the 5110 again accepts input from the keyboard.

   Note: Using the)PROC command requires approximately 800 bytes of active workspace storage. The active workspace is always cleared the first time a)PROC command is used after you turn the power on or press RESTART, because the required 800 bytes can be allocated only in a CLEAR WS. The active workspace is not cleared for any subsequent)PROC commands because the 800 bytes are already allocated until the power is turned off or RESTART is pressed. Therefore, if you have any useful data in the active workspace, you should save this data before using the)PROC command the first time.

INFORMATION PROVIDED ABOUT THE ACTIVE WORKSPACE

You can get information about the active workspace by simply entering certain system commands, system variables, or system functions without specifying any parameters or arguments. The system commands that provide information about the active workspace are:

System Command	*Information Provided*
)SYMBOLS	How many symbols are allowed and how many symbols are currently being used. (The symbols consist of labels, variable names, user-defined function names, and any system variables and functions that have been used.)
)WSID	The current workspace ID and device/file number. This information specifies where the active workspace is stored when a)SAVE or)CONTINUE command is used. (When a stored workspace is loaded into the active workspace, the workspace ID and device/file number of the stored workspace is assigned to the active workspace.)
)FNS	The name of the user-defined functions in the active workspace.
)VARS	The names of the variables in the active workspace.
)SI	The names of any suspended functions and the statement number where each function is suspended. For example:

```
      )SI
FUNCTIONA[5] *
FUNCTIONB[2]
FUNCTIONC[6] *
```

Statement 2 of this function called FUNCTIONA.

Statement Number

Function Name

The * (asterisk) indicates the *suspended* functions. The functions without an asterisk are functions (called *pendent* functions) that called the previous function.

)SINL	The same information as the)SI command plus the names local to each function.

The system variables that provide information about the active workspace are:

System Variable	*Information Provided*
⎕CT	How different two numbers must be to be considered unequal.
⎕IO	The value of the index origin.
⎕RL	The starting value used in generating random numbers.
⎕LC	The statement number currently active. These statement numbers are the same as the statement numbers displayed by the)SI command.
⎕WA	The amount of unused storage in the active workspace.

Note: The value of these system variables can be used in APL statements. For example:

→⎕LC causes a suspended function to resume execution with the next statement to be executed.

The system functions that provide information about the system are:

System Function	*Information Provided*
⎕NL	The names of the labels, variables, and user-defined functions in the active workspace
⎕NC	The classification (label, variable, or user-defined function) of a specified name

ACTIVE WORKSPACE STORAGE CONSIDERATIONS

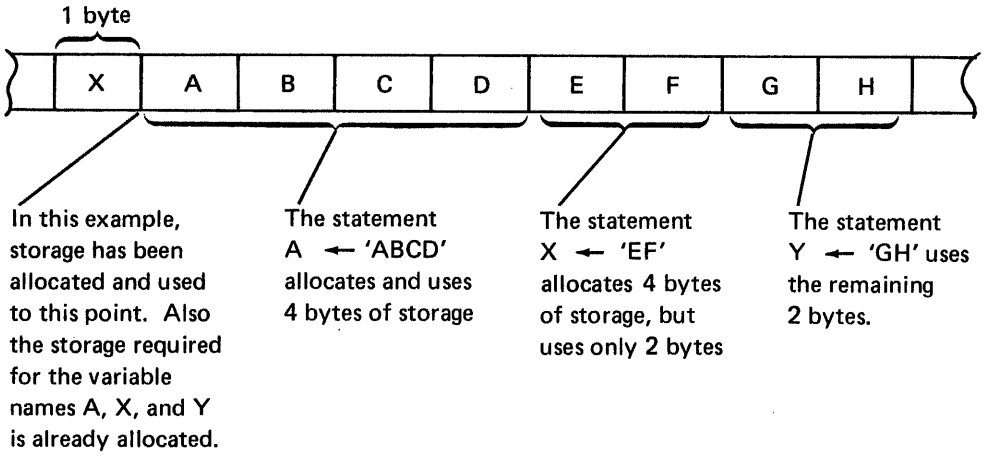
Because the 5110 active workspace contains a fixed amount of storage, it is good practice to conserve as much storage as possible.

Data Types and Storage Considerations

The following list shows how many bytes of storage are required for each data type that can be in the active workspace:

Data Type	*Number of Bytes Required*
Character constant	1 byte per character
Variable name (3 characters or less)	12 + (4 x rank) bytes The rank is the number of coordinates ($\rho\rho$variable).
Variable name (4 characters or more)	1 byte per character + 21 bytes + (4 x rank) bytes
Whole numbers that are equal to or less than 2^{31} -1	4 bytes
Whole numbers that are greater than 2^{31} -1	8 bytes
Decimal numbers	8 bytes
Logical data	1 bit (1 byte contains 8 one or zero bits)

Note: Storage is always allocated in 4 byte increments. For example, the following illustration represents a portion of storage:



Following are some considerations that can be used to conserve storage:

- Make all objects (variables and user-defined functions) not required for use outside of a user-defined function local to the function.

- Store data in data files on tape or diskette and use an APL shared variable (see *Input/Output Control*) to transfer the data into the active workspace when required.

- Clear suspended functions from the active workspace.

- Collect user-defined functions by related operations and store each set into a workspace file on the media. Then when a certain set of related functions is required to process data in the active workspace, the stored workspace containing these functions can be copied into the active workspace. When the processing is done, the functions can be expunged (⎕EX) and another set of functions can be copied into the active workspace.

- If a value consists of all 1's and 0's, store the value as logical data. For example, you have the following vector:

```
      VECTOR←10⍴(2-1)
      VECTOR
1 1 1 1 1 1 1 1 1 1
```

  The result is a vector of ten 1's, and each 1 requires 4 bytes of storage. However, the vector can be changed to a logical vector as follows:

```
      VECTOR←1∧VECTOR
      VECTOR
1 1 1 1 1 1 1 1 1 1
```

The result looks just like the previous result; however, only 2 bytes of storage are required.

- Because each variable requires at least 12 bytes of overhead, an array of six elements requires approximately 60 bytes less storage than six separate variables.

- Names of 3 characters or less require 8 bytes of storage in the symbol table (the symbol table is part of the active workspace where the names of all the symbols, including variables, user-defined functions, and labels, are stored). Names of 4 characters or more require an additional 8 bytes plus 1 byte for each character in the name.

  Note: Even if an object is erased from the active workspace, the storage used for its name will not be available for use unless the contents of the active workspace are written to the media with a)SAVE command and then loaded or copied back into the active workspace.

- Identical names that are local to more than one user-defined function do not require additional symbol-table space for each function.

When the contents of the active workspace are written to the media by the)CONTINUE command, and that stored workspace is loaded into a different 5110 with a larger active workspace, the amount of available workspace (□WA) remains the same as it was when the contents of the active workspace were originally written to the media. To take advantage of the additional storage in the larger active workspace, write the contents of the active workspace to the media using the)SAVE command, then load the stored workspace back into the 5110. Also, for the same reason, a workspace written to the media by the)CONTINUE command cannot be loaded into a different 5110 with a smaller active workspace.

The following formula shows how much storage in the active workspace is required to perform an input or output operation to tape or diskette using an APL shared variable (see *Input/Output Control*):

REQUIRED STORAGE = BUFFER + SHARED VARIABLE + OVERHEAD

where:

- *REQUIRED STORAGE* is the amount of storage that must be available in the active workspace (see ⎕WA) before an input or output operation to tape or diskette can be performed. If there is not enough available storage, a WS FULL error occurs.

- *BUFFER* is the amount of storage required by the data assigned to the shared variable. This storage is only used during the input/output operation.

- *SHARED VARIABLE* is the amount of storage required for the data assigned to the shared variable.

- *OVERHEAD* is the amount of storage used when the input/output operation is specified. The overhead is as follows:

  OUT and OUTF operations–812 bytes
  IN operation–792 bytes
  INR, IOR, IORH operations–356 bytes

ADDITIONAL STORAGE USING DISKETTE DATA FILES

You can use a direct access data file to store variables that are not currently needed in the active workspace. These variables can then be removed from the active workspace (to conserve storage) and quickly read back into the active workspace as needed. You can use the following procedure to store variables for later use on a diskette data file:

1. Establish a type M direct access data file (see *Input/Output Control*) using dummy records. The first dummy record written to the file should require as much diskette storage as the largest variable that is to be written to the file. For example, you might use the following user-defined function to establish a type M file with 256 bytes allocated for each record:

```
      ∇DUMMY[⎕]∇
    ∇ DUMMY;C;X;I
[1]   I←0
[2]   X←1 ⎕SVO 'C'
[3]   C←X←'OUTF 11008 ID=(REAL.STORAGE) TYPE=M'
[4]   →(0≠1↑X←C)/ERROR
[5] LOOP:C←256⍴'A'
[6]   →(0≠1↑X←C)/ERROR
[7]   →(20≥I←I+1)/LOOP
[8]   C←⍳0
[9]   →(0≠1↑X←C)/ERROR
[10]  'THE DATA FILE IS CREATED SUCCESSFULLY'
[11]  →0
[12] ERROR:'ERROR--THE RETURN CODE IS: ',⍕X
    ∇
```

You must specify the file ID enclosed in single quotes.

2. Establish a pair of shared variables, and specify direct access input/output operations to the data file. For example, you might use the following user-defined function:

```
      ∇OPEN[⎕]∇
    ∇ OPEN FILEID;X
[1]   X←1 ⎕SVO 2 7 ⍴'CTLSAVEDATSAVE'
[2]   →(∧/X≠2)/ERROR
[3]   CTLSAVE←'IOR ID=(',FILEID,')'
[4]   X←CTLSAVE
[5]   →(0≠1↑X)/ERROR2
[6]   'SUCCESSFUL OPERATION'
[7]   SAVELIST← 1 1 ⍴' '
[8]   →0
[9] ERROR:'THE OFFER FAILED--THE RESULT IS: ',⍕X
[10]  →0
[11] ERROR2:'THE OPEN FAILED--THE RETURN CODE: ',⍕X
    ∇
```

You must specify the file ID enclosed in single quotes.

3. Store the variable in the data file. For example, you might use the following user-defined functions:

The left argument must be 1 to expunge the variable name from the active workspace; otherwise, the left argument must be 0.

The right argument is the variable name enclosed in single quotes.

```
      ∇SAVE[⎕]∇
    ∇ EXPUNGE SAVE VAR;MASK;X
[1]   ⍎(0=⎕NC 'SAVELIST')/'SAVELIST←0 0⍴'' '''
[2]   ⍎((⍴SAVELIST)[1]<⍴VAR←,VAR)/'SAVELIST←SAVELIST BY VAR'
[3]   ⍎(~∨/MASK←(((⍴SAVELIST)[1]↑VAR)∧.=SAVELIST))/'SAVELIST ←
      SAVELIST BY VAR'
[4]   NL←(⍴SAVELIST)[1]
[5]   DATSAVE←⍎VAR
[6]   CTLSAVE←1,((NL↑VAR)∧.=SAVELIST)\1
[7]   →(0≠+/CTLSAVE)/ERROR
[8]   X←0
[9]   ⍎EXPUNGE/'X←⎕EX ''',VAR,''''
[10]  'THE VARIABLE IS SAVED ',(X/'AND EXPUNGED')
[11]  →0
[12] ERROR:'SAVE FAILED'
    ∇
```

```
      ∇BY[⎕]∇
    ∇ Z←A BY B;ROW
[1]   A←(2↑(⍴A),1 1)⍴A
[2]   B←(2↑(⍴B),1 1)⍴B
[3]   ROW←1↑(⍴A)⌈⍴B
[4]   A←(ROW,¯1↑⍴A)↑A
[5]   B←(ROW,¯1↑⍴B)↑B
[6]   Z←A,B
    ∇
```

This function is used by the SAVE function.

4. At a later time, read the stored variable back into the active workspace. For example, you might use the following user-defined function:

The right argument is the variable name enclosed in single quotes.

```
      ∇FETCH[⎕]∇
    ∇ Z←FETCH VAR;MASK;NL
[1]   →(0=⎕NC 'SAVELIST')/NOTFOUND
[2]   →((⍴VAR←,VAR)>NL←1↑⍴SAVELIST)/NOTFOUND
[3]   →(~∨/MASK←(NL↑VAR)∧.=SAVELIST)/NOTFOUND
[4]   CTLSAVE←0,MASK\1
[5]   →(0≠+/CTLSAVE)/0
[6]   Z←VAR,' IS FOUND'
[7]   ⍎VAR,'←DATSAVE'
[8]   →0,DATSAVE←\0
[9]  NOTFOUND:Z←VAR,' IS NOT FOUND IN THE FILE'
    ∇
```

5. Before the contents of the active workspace can be written to the media by the)SAVE command, the input/output operations to the data file must be terminated. For example, you might use the following user-defined function:

```
     ∇CLOSE[[]]∇
    ∇ CLOSE FILEID                    ← You must specify the file ID
[1]   CTLSAVE←⍳0                        enclosed in single quotes.
[2]   X←CTLSAVE
[3]   →(0≠1↑X)/ERROR
[4]   'THE FILE CLOSED SUCCESSFULLY'
[5]   →0
[6]  ERROR:'THE FILE DID NOT CLOSE--THE RETURN CODE IS:'
[7]   X
    ∇
```

Note: If you want to use the data file at a later time, do not terminate the input/output operations. Instead, you should use the)CONTINUE command to write the contents of the active workspace to the media and then use the)RESUME command to reestablish the input/output operations at a later time.

This procedure works as follows, using the sample user-defined functions:

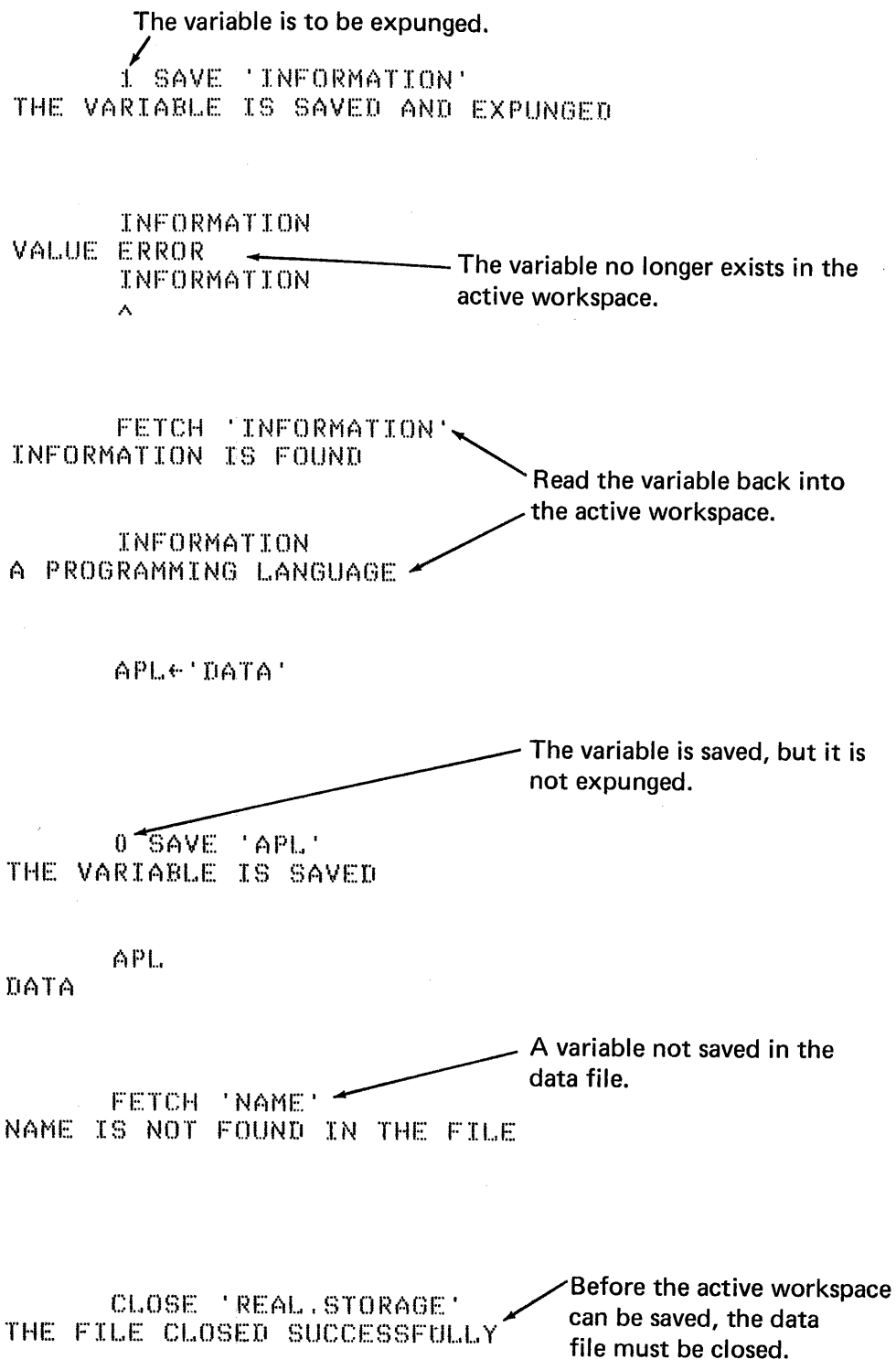
```
      )MARK 10 1 8 11                 ← File number 8 is used as
MARKED 0008 0010                        the data file.
```

```
      DUMMY                           ← Create the data file
THE DATA FILE IS CREATED SUCCESSFULLY   used to save the
                                        variables.
```

```
      OPEN 'REAL.STORAGE'             ← The data file is ready
SUCCESSFUL OPERATION                    for input/output operations.
```

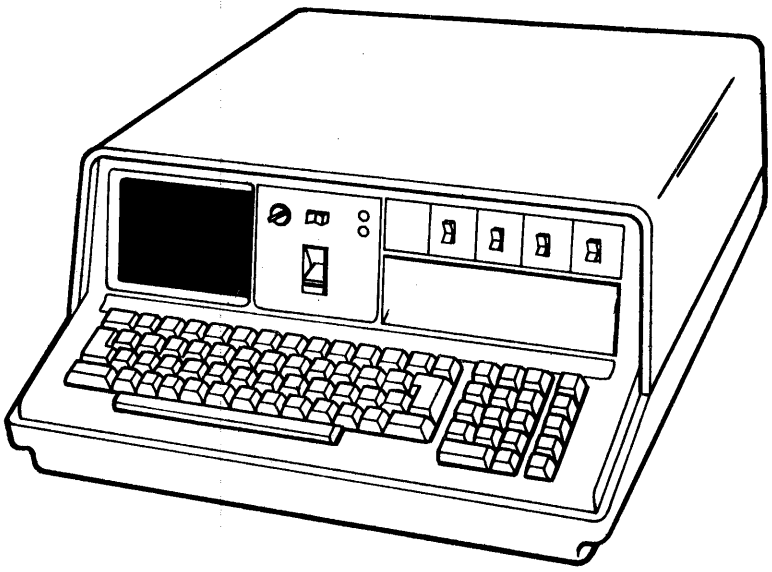
A variable to be saved for later use.

```
INFORMATION←'A PROGRAMMING LANGUAGE'
```


The variable is to be expunged.
1 SAVE 'INFORMATION'
THE VARIABLE IS SAVED AND EXPUNGED
INFORMATION
VALUE ERROR
INFORMATION
^
The variable no longer exists in the active workspace.
FETCH 'INFORMATION'
INFORMATION IS FOUND
Read the variable back into the active workspace.
INFORMATION
A PROGRAMMING LANGUAGE
APL←'DATA'
The variable is saved, but it is not expunged.
0 SAVE 'APL'
THE VARIABLE IS SAVED
APL
DATA
A variable not saved in the data file.
FETCH 'NAME'
NAME IS NOT FOUND IN THE FILE
CLOSE 'REAL.STORAGE'
THE FILE CLOSED SUCCESSFULLY
Before the active workspace can be saved, the data file must be closed.

Chapter 5. Console Control

The 5110 console consists of a keyboard, a display screen, and switches.

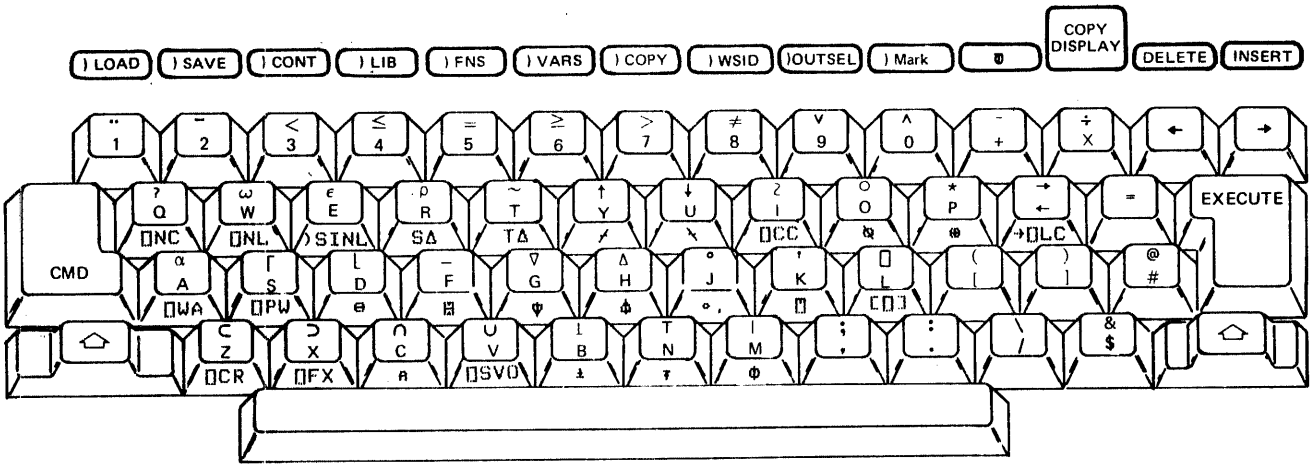


In this chapter, the following topics concerning the 5110 console are discussed:

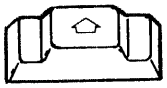
- Controlling the input from the keyboard
- Controlling the position of the information on the display screen
- Sounding the audible alarm
- Console control through a user-defined function

CONTROLLING THE INPUT FROM THE KEYBOARD

The following illustration shows the positions on the alphameric keyboard of the uppercase alphameric characters, the APL symbols, the APL keywords, and the special character combinations:

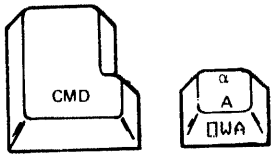


Note: The special character combinations are engraved on the front of the appropriate key on the 5110 keyboard. If the 5110 is a combination APL/BASIC machine, the special APL character combinations are below the BASIC keywords.

When you turn the 5110 power on, the 5110 is in standard APL character mode. That is, you enter the uppercase alphabetic characters without using the shift  key, the APL symbols using the shift key, and the APL keywords and special character combinations using the CMD key. For example, if you press:

The character A is entered from the keyboard.

The character α is entered from the keyboard.



The character combination □WA is entered from the keyboard.

You can also enter lowercase alphabetic characters from the keyboard. To do this, you use one of the following procedures:

- Use the keying sequence:
  1. Press the HOLD key.
  2. Hold down the (shift) key and press the (scroll down) key.

- Enter the following statement to use the console control (⎕CC) system function:

A 3 as the left argument specifies that the ⎕CC system function is used to change the character mode.

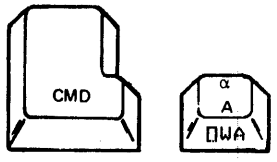
```
      3 ⎕CC 1
1
```

Specifies lowercase alphabetic characters. (See the *IBM 5110 APL Reference Manual*, SA21-9303, for a complete description of the ⎕CC system function.)

Once the 5110 is in lowercase character mode, you enter the lowercase alphabetic characters without using the shift key, the uppercase alphabetic characters using the shift key, and APL symbols using the CMD key. For example, if you press:

The character a is entered from the keyboard.

The character A is entered from the keyboard.



The character α is entered from the keyboard.

You cannot enter the special character combinations using the CMD key and a single key when the 5110 is in lowercase character mode.

The 5110 remains in lowercase character mode until:

- The system power is turned off.
- The RESTART switch is pressed.
- One of the following procedures is used to change the keyboard entry mode to standard APL characters:
  -- Perform the keying sequence:
     1. Press the HOLD key.
     2. Hold down the (shift) key and press the (scroll up) key.
  -- Enter the statement:

```
      3 ⎕CC 0
1
```

Specifies standard APL character mode

CONTROLLING THE INFORMATION ON THE DISPLAY SCREEN

You can control the information and the position of the information on the display screen by using:

- The scroll up and scroll down keys
- The ⎕CC (console control) system function to turn display on or off
- The ⎕CC system function to scroll the information on the display screen up or down
- A pair of shared variables

You are already familiar with using the scroll up and scroll down keys to position the information on the display screen. How to use a pair of shared variables to read and write data at any position on the display screen is discussed in detail under *Input/Output Control*. How to use the ⎕CC system function to control information on the display is discussed in this section.

Using the □CC System Function to Control on the Display Screen

In the previous discussion on controlling the input from the keyboard, the □CC function was used to place the 5110 in lowercase alphabetic character mode. The □CC function can also be used to:

- Turn the display screen on or off during the execution of a user-defined function. The primary advantage in turning the display screen off is that the 5110 internal processing speed is increased by approximately 18%.
- Scroll the information on the display screen up or down.

To turn the display screen off during the execution of a user-defined function, the left argument of the □CC function must be a 1 and the right argument must be 0. The display screen remains off until one of the following occurs:

- The user-defined function completes execution.
- The □CC function is used to turn the display screen on again (both the left and right arguments are specified as 1).
- A later statement in the user-defined function generates a result to be displayed.

Since the □CC function generates an explicit result, that result must be assigned to a variable to prevent the display screen from being immediately turned on and the result displayed. For example:

```
[1]   R←1 □CC 0
```

When the □CC function is used to scroll the display screen up or down, the left argument must be 4 and the right argument specifies the number of lines to scroll up or down. For example:

`R←4 □CC 8`	Scrolls the information on the display screen *up* 8 lines
`R←4 □CC ¯3`	Scrolls the information on the display screen *down* 3 lines

In the previous examples, the explicit result of the □CC function is assigned to a variable. Otherwise, the explicit result is displayed after the □CC function scrolls the display screen up or down.

SOUNDING THE AUDIBLE ALARM

Another use of the ⎕CC function is to sound the audible alarm feature, if installed. To do this, the left argument must be a 2 and the right argument is:

- A 1 to sound the audible alarm. The audible alarm remains on until any input is entered from the keyboard or the ⎕CC function is used to turn the audible alarm off (the right argument is a 0). For example:

```
R←2 ⎕CC 1 0 1 0 1 0
```

  turns the audible alarm on and off three times. The on/off interval is approximately 0.006 second.

- A vector of 2's, where each 2 sounds the audible alarm for approximately 1/4 second. For example:

```
R←2 ⎕CC 2 2 2 2
```

  sounds the audible alarm four times for approximately 1/4 second each time.

CONSOLE CONTROL THROUGH A USER-DEFINED FUNCTION

The ⎕CC system function, like any other system function, can be executed from a user-defined function. This technique allows the following console control operations to be initiated from a user-defined function:

- Turn the display screen on or off.
- Sound the audible alarm.
- Set the keyboard input to standard APL characters or lowercase alphabetic characters.
- Scroll the display screen up or down.
- Set the left tab position for printed output (see *Printer Control* for more information on setting the left tab position).

Following is an example of initiating console control operations from a user-defined function:

```
      ∇CONTROL[⎕]∇
    ∇ CONTROL;NAME;STATEMENT;R
[1]   R←4 ⎕CC 16
[2]   R←3 ⎕CC 1
[3]   'ENTER YOUR FIRST AND LAST NAME'
[4]   NAME←⍞
[5]   R←3 ⎕CC 0
[6]   R←4 ⎕CC 10
[7]   ⍞←'WELCOME '
[8]   ⍞←NAME
[9]   ', ENTER A STATEMENT TO BE EXECUTED.'
[10]  R←4 ⎕CC 2
[11]  STATEMENT←⍞
[12]  R←4 ⎕CC 2
[13]  R←1 ⎕CC 0
[14]  R←⍎STATEMENT
[15]  'THE RESULT IS: '
[16]  R
[17]  R←2 ⎕CC 2 2 2 2
    ∇
```

Scroll the display screen up 16 lines.

Set the keyboard input to lowercase alphabetic character mode.

Set the keyboard input to standard APL character mode.

Turn the display screen off to improve internal performance when the next statement is executed.

The display screen is turned on again when output is sent to the display screen.

Sound the audible alarm when the user-defined function has completed execution.

Chapter 6. Library Control

The tape or diskette files where you store information is your *library*. In this chapter, the following topics concerning the 5110 library control are discussed:

- Determining the size of a tape or diskette file
- Writing data to a tape or diskette file
- Getting data from a tape or diskette file
- Controlling files in the library
- Maintaining data security

References are made to some of the 5110 system commands; for example, the)MARK command or the)RESUME command. See the *IBM 5110 APL Reference Manual*, SA21-9303, for a complete description of the system commands. The description includes the required syntax for each system command. You must use the proper syntax to enter a system command so that the 5110 will accept that command.

DETERMINING THE SIZE OF A TAPE OR DISKETTE FILE

Before information can be stored on the media (tape or diskette), the media files must be formatted by the)MARK command. When using the)MARK command, you can use the following formulas to determine the maximum size a file should be marked. The formula for a workspace file [the contents of the active workspace were written to the storage media with a)SAVE or)CONTINUE command] is MAXSIZE = ⌈3 + (CLEAR-ACTIVE) ÷ 1024, where:

- MAXSIZE is the maximum amount of media storage (number of 1024-byte blocks) that would be required to write the contents of the active workspace to the media.
- CLEAR is the value of ⎕WA in a clear workspace.
- ACTIVE is the value of ⎕WA just before the contents of the active workspace are written to the media.

The formula for a data file (data written to the media using an APL shared variable) when all of the data is contained in the active workspace is MAXSIZE = ⌈ (WITHOUT-WITH) ÷ 1024, where:

- MAXSIZE is the maximum amount of media storage (number of 1024-byte blocks) required to write the data to the media.
- WITH is the value of ⎕WA with the data in the active workspace.
- WITHOUT is the value of ⎕WA before any data to be written to tape or diskette was stored in the active workspace.

There is no formula for determining what size to mark a data file when the data is written to the media as it is entered from the keyboard. The amount of storage required depends upon how much data is entered from the keyboard and what type of data is used. For information on how many bytes of storage are required by the various types of data, see *Storage Considerations* in the *Active Workspace Control* chapter.

WRITING DATA TO A TAPE OR DISKETTE FILE

You can write data to a file by using the)SAVE command, the)CONTINUE command, or an APL shared variable. The)SAVE or)CONTINUE commands are used to write the contents of the 5110 active workspace to a file. An APL shared variable is used to write individual data records to a data file (shared variables are discussed in detail under *Input/Output Control*).

The following list shows the advantages and disadvantages of using the)SAVE command versus the)CONTINUE command. You should consider these advantages and disadvantages when choosing which command you are going to use when writing the contents of the active workspace to the media.

Using the)SAVE Command

Advantages:

- The stored workspace can be loaded into a 5110 that has a smaller active workspace than the original active workspace, providing the stored workspace does not require more storage than is available in the smaller active workspace.
- The additional storage is available to the user when the stored workspace is loaded into a 5110 that has a larger active workspace than the original active workspace.
- The symbol table is cleared of unused or expunged symbol references.
- The)COPY and)PCOPY commands can be used to copy specified variables or user-defined functions from the stored workspace.

Disadvantages:

- The active workspace is not written to the media as fast as when the)CONTINUE command is used.
- The stored workspace is not loaded into the active workspace as fast as when the)CONTINUE command is used to write the workspace to the media.
- If the active workspace contains suspended functions or open data files exist, the active workspace cannot be written to the media.

Advantages:

- Even if the active workspace contains suspended functions or open data files exist, the active workspace can be written to the media. Any open data files or suspended functions can be restored by the)RESUME command. The)RESUME command is discussed in more detail under *Getting Data from a Tape or Diskette File* in this section.

- The active workspace is written to the media faster than when the)SAVE command is used.

- The stored workspace is loaded into the active workspace faster than when the)SAVE command is used to write the workspace to the media.

Disadvantages:

- The)COPY and)PCOPY command cannot be used to copy variables or user-defined functions from the stored workspace.

- The stored workspace can only be loaded into a 5110 with an active workspace at least as large as the original active workspace.

- The additional storage is *not* available to the user when the stored workspace is loaded into a 5110 that has a larger active workspace than the original active workspace.

GETTING INFORMATION FROM A TAPE OR DISKETTE FILE

You can read information from a tape or diskette file by using the)LOAD,)RESUME,)COPY,)PCOPY commands, or an APL shared variable. The)LOAD,)COPY,)PCOPY, and)RESUME commands are used to place the contents of a stored workspace into the 5110 active workspace. An APL shared variable is used to read individual data records from a data file (shared variables are discussed in detail under *Input/Output Control*).

Generally, you use the)LOAD command to replace the contents of the active workspace with the contents of a stored workspace. However, if there were shared variables or suspended functions in the active workspace and the)CONTINUE command was used to write the active workspace to the media, the)RESUME command reads the stored workspace into the active workspace and reestablishes the shared variables and/or suspended functions. That is, the system environment is reestablished as it was when the)CONTINUE command was issued.

Using the)CONTINUE and)RESUME commands allows you to work with shared variables or suspended functions, write the active workspace to the media, and then reestablish the system environment at a later time so that you can continue working with the shared variables and/or suspended functions. See *The)RESUME Command* in the *IBM 5110 APL Reference Manual*, SA21-9303, for a description of how the shared variables are reestablished.

If a workspace was stored on the media using the)SAVE command, you can copy all or part of that workspace into the active workspace by using the)COPY or)PCOPY commands. The)COPY command copies all or specified objects (variables or user-defined functions) into the active workspace and replaces any objects in the active workspace that have the same name. The)PCOPY command copies all or specified objects into the active workspace; however, any objects in the active workspace that have the same name are not replaced (they are protected). The)COPY and)PCOPY commands allow you to read more than one stored workspace or parts of more than one stored workspace into the active workspace without replacing objects already existing in the active workspace.

CONTROLLING THE FILES IN THE LIBRARY

Once you have stored several workspace and data files on a tape or diskette (your library), you might want to know what files you have in your library. You can use the)LIB command to display the file headers for a specified tape or diskette. The file headers provide you with such information as the file number, the file ID, the file type, and so on. See *The)LIB Command* in the *IBM 5110 APL Reference Manual*, SA21-9303, for a description of the information contained in the file header.

When there are files on tape or diskette that contain data that is no longer required, you can mark these files unused by issuing the)DROP command. Once a file is marked unused, any data in the file can no longer be read into the 5110 and the file is available for other uses.

Also, if a diskette file is no longer required, you can make the file space available for reallocation by issuing the)FREE command. This allows the file space on the diskette to be used for other numbered files by the)MARK command. See *Diskette Concepts* for more information on how files are allocated on a diskette.

DATA SECURITY

You are primarily responsible for the security of any sensitive data. After you are through using the 5110, you can remove the data in the active workspace by one of the following:

- Using the)CLEAR command to clear the active workspace
- Pressing the RESTART switch
- Turning the POWER ON/OFF switch to off

There are several methods available for protecting or removing sensitive data on a tape or diskette. These methods are:

- Assigning a password to the workspace when the system is writing the active workspace on the media.
- Rewriting a file, which makes the old data inaccessible.
- Filling a data file with meaningless data. For example, the following user-defined function fills file 4, a data file named DATA on tape 1, with zeros:

```
      ∇SECURITY;A;B
[1] 1 ⎕SVO 'A'
[2] A←'OUT 1004 ID=(DATA)'
[3] B←10 1000⍴0
[4] WR:A←B
[5] →(A[1]=0)/WR∇
```

- Setting the tape cartridge SAFE switch in the SAFE position to prevent someone from accidently writing on the tape.
- Using the)PROTECT command to prevent someone from accidently writing on a diskette file.
- Using the)VOLID command to prevent unauthorized access to the diskette files.
- Storing the tape or diskette in a secure place.

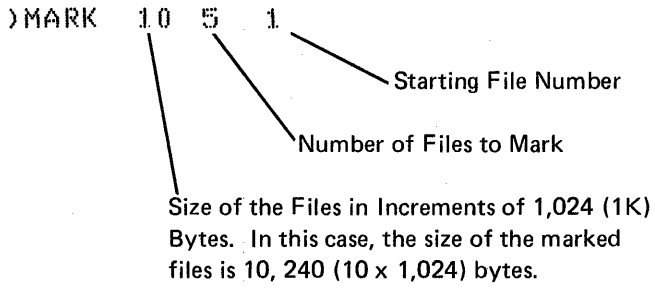
Chapter 7. Tape Concepts

There are 204K bytes (1K = 1024 bytes) of tape storage available on an IBM Data Cartridge. This tape storage is used for file headers, workspace files, and data files. In this section, the following topics are discussed:

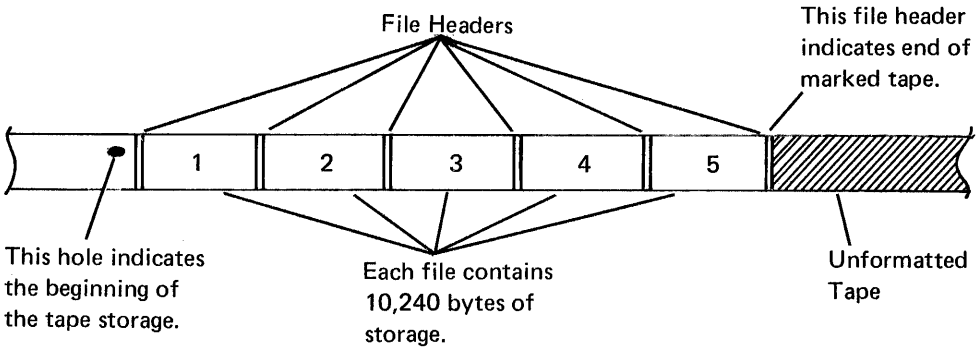
- How to format the tape
- How to determine the amount of storage on a tape cartridge that is actually available to you

FORMATTING THE TAPE

You must use the)MARK command to format files on the tape before you can store the contents of the active workspace or data records on the tape. For example:

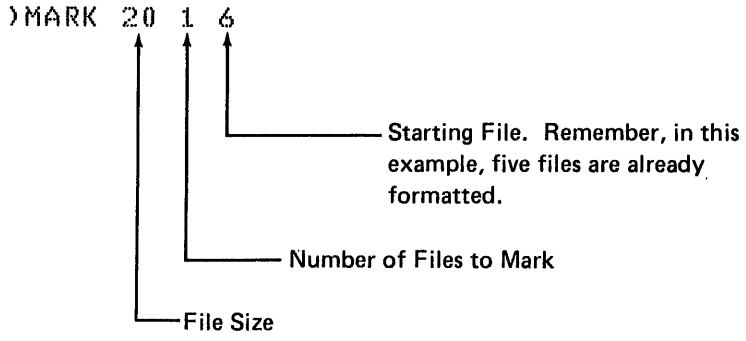


Once the previous)MARK command is successfully completed by the 5110, the tape is formatted as follows:

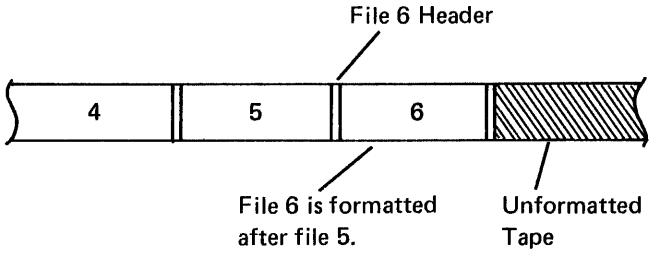


The file headers contain information about the file, such as the file number, file name, file type, and so on. Each file header requires 512 bytes of tape storage.

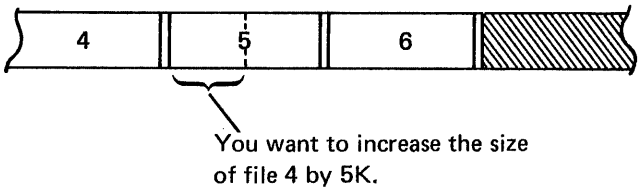
Now, if you want to format additional files on the tape, you must use the)MARK command again. For example:



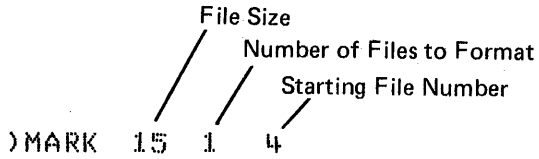
The tape is now formatted as follows:



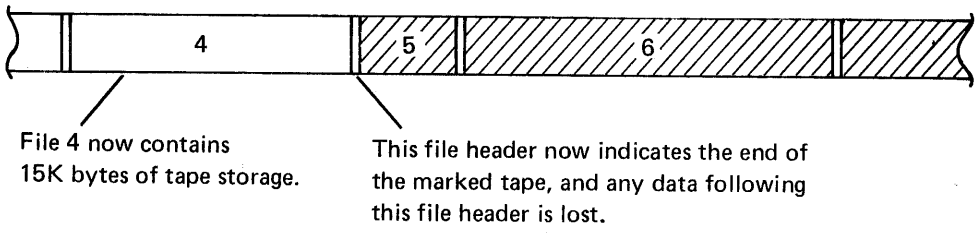
When the information in a tape file is no longer needed, you can use the)DROP command to mark the file unused so that the file is available for other uses. However, once a file is formatted, you cannot increase the size of the file without re-marking the file. When you re-mark an existing file, any information in the files following the re-marked file is lost. For example, assume you want to increase the size of file 4 on the tape from 10K to 15K:



After the command:



is successfully completed, the tape is formatted as follows:



A formatted tape has the following characteristics:

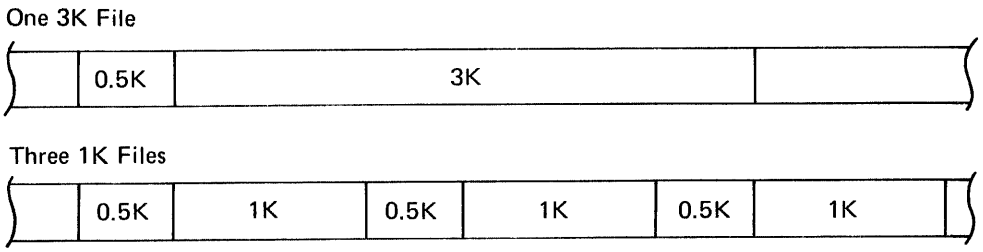
- The files are of variable length from 1K to 200K, in 1K increments.
- The files can be randomly accessed; that is, you can read a file without having to read the previous file. However, the data in the files must be accessed sequentially.
- It can contain both workspace and data files.
- It can contain both APL and BASIC files.

DETERMINING THE AMOUNT OF STORAGE AVAILABLE ON A TAPE CARTRIDGE

There are approximately 204K bytes of storage on each tape cartridge, but the amount of tape storage actually available to you depends on:

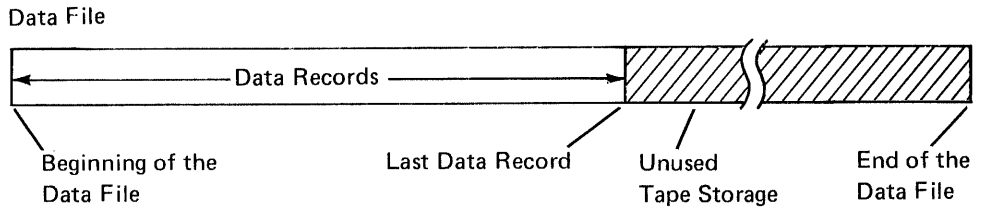
- How many files are marked (formatted) on the tape
- How the data files were written to tape

Each file on a tape cartridge requires one 512-byte file header. Therefore, as you mark more files on a tape cartridge, more tape storage is used for file headers. For example, if you mark one 3K file on a tape, 512 bytes of tape storage are used for the file header. However, if you mark three 1K files on tape, 1,536 bytes of tape storage are required for the file headers.

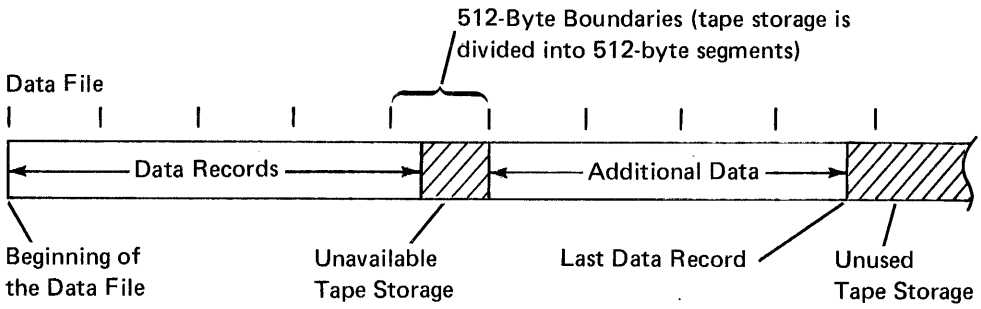


Notice that, in each case, a total of 3K bytes of tape storage is allocated for tape files. However, in the second case, an additional 1K bytes more of tape storage are used.

The amount of data you can store in a data file depends on how the data is written to the data file. (See *Input/Output Control* for a complete description of writing data to data files.) For example, when you first write data to a data file (an OUT operation), the individual records are sequentially written to tape starting at the beginning of the data file. Once these records are written to tape, the data file might look like this:



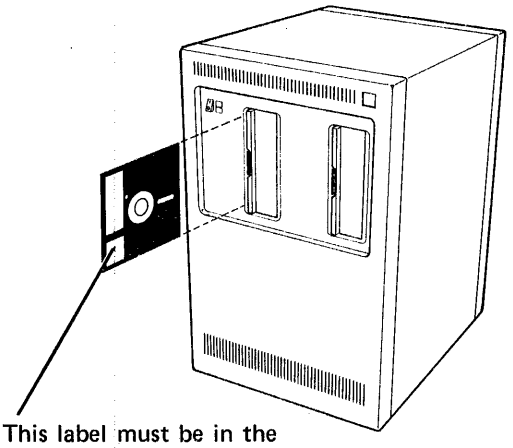
Now, assume you add data to the data file at a later time (an ADD operation). The new data starts at the first 512-byte boundary after the last record in the data file. The tape storage between the last data record and the additional data records is unavailable for use. Once the new data records are written to tape, the data file might look like this:



As you add more data to the file, it is possible for more tape storage to become unavailable. You can *compress* the data in the data file (use the unavailable storage) by first reading all the data records from the file and then writing the data records back to the file, starting at the beginning of the data file (an OUT operation).

Chapter 8. Diskette Concepts

The IBM diskette is a thin, flexible disk permanently enclosed in a semirigid, protective, plastic jacket. When the diskette is properly inserted in the diskette drive, the disk turns freely within the jacket. The diskette is inserted in the diskette drive as follows:



This label must be in the lower corner as the diskette is inserted in the diskette drive. The diskette drive door must be closed and latched after the diskette is inserted.

Data is written on the diskette at specific locations (addresses) by the system. These addresses provide direct access to specific information. Data written at an address remains there until it has been replaced by new data. To read data, the desired address is found and the data is read into the 5110.

Before being shipped to a user, each IBM-supplied diskette is initialized. Initialization is a process whereby label information and data addresses are recorded on the diskette. In this chapter, the following topics are discussed:

- Diskette wear
- Diskette addressing and layout
- Diskette types and formats
- Diskette initialization

- Diskette volume ID, owner ID, and access-protect indicator
- Diskette file information, such as file ID, write-protect indicator, and organization
- Reallocation of diskette file space
- Amount of storage available on a diskette

DISKETTE WEAR

The use of flexible diskette storage provides some significant advantages, such as low cost, compact size, multiple system functions, and ease of media handling and storage. It should be recognized, however, that during recording and reading, the read/write head is in contact with the media, causing diskette wear over time. Variations in the rate of wear will depend on the particular operating environment and application characteristics. Care in the storage, use, and handling can also affect diskette life. (See guidelines in the *IBM 5110 APL Reference Maual*.) Excessive wear, handling, or contamination can cause possible failures in recording and/or reading.

Ultimate wear is to some extent dependent upon total usage of individual tracks. Care taken to distribute data so that accessing occurs over the entire recording surface with about the same frequency can extend the useful life of the diskette. Actual experience with individual applications and environments will allow development of guidelines as to when the media should be replaced.

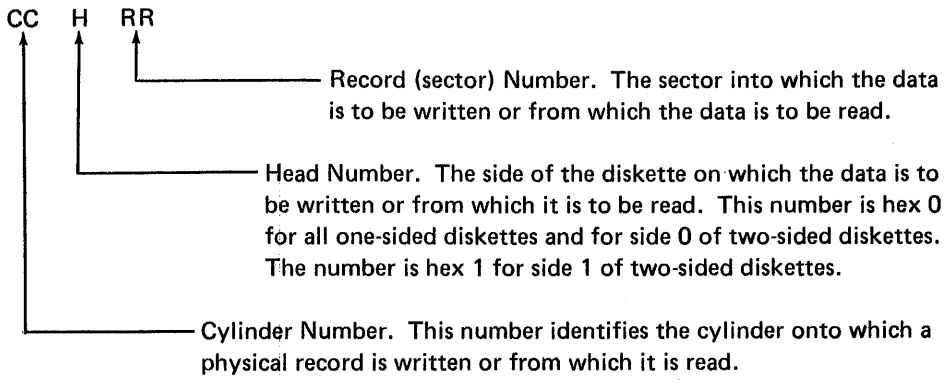
Unpredictable circumstances such as contamination or severe handling can cause an early error to occur.

For the previous reasons, consideration should be given to providing an adequate recovery plan, such as:

- Backing up critical programs and data files on a second diskette for use in the event of an error on the primary diskette.
- Periodically moving frequently used files to alternate locations on the diskette (see the copy function in the *IBM 5110 Customer Support Functions Reference Manual*).

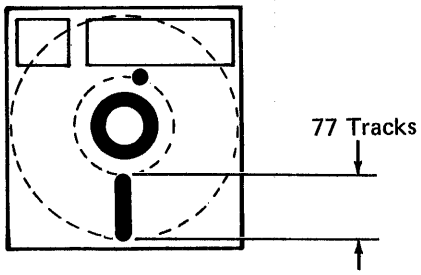
DISKETTE ADDRESSING AND LAYOUT

A diskette address consists of a combination of cylinder number, head number, and record number as follows:



Track and Cylinder

A track is the recording area on a single diskette that passes the read/write head while the disk makes a complete revolution. The read/write head is held by a carriage that can be moved to 77 distinct locations along a straight line from the center of the disk. Therefore, each diskette has 77 concentric tracks on which data can be stored.

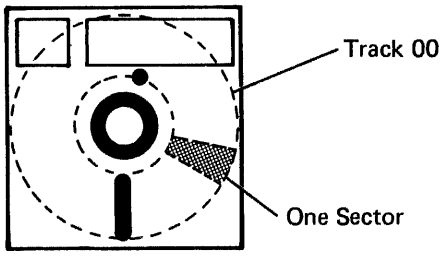


The diskette drive for two-sided diskettes has a read/write head on each side. Each track on side 0 of a two-sided diskette has an associated track on side 1.

A cylinder is one track on a one-sided diskette or a pair of associated tracks (the corresponding tracks on opposite sides of the diskette) on a two-sided diskette. There are 77 cylinders (numbered 0 to 76) on a diskette.

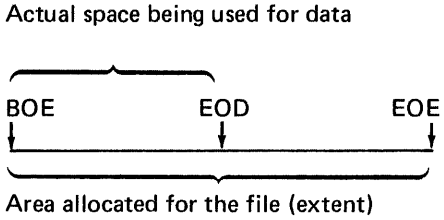
Sector

A sector is a portion of a cylinder. All sectors on a single cylinder are the same size, and the number of sectors on a cylinder depends on the number of bytes per sector (see *Diskette Types and Formats* in this chapter).



Index Cylinder

Cylinder 0 is called the index cylinder and is reserved for information describing the diskette and its contents. It contains information about the diskette, such as volume and owner identification. The index cylinder also contains information associated with each file on the diskette. This includes the name of each file and the addresses associated with the file extents. An extent is the maximum space a file can occupy. The address at the beginning of this space is called the beginning of extent (BOE). The address at the end of this space is called the end of extent (EOE). A file may not use all of the space allocated for it by the BOE and EOE addresses; therefore, another address for end of data (EOD) exists.



The EOD address is used to identify the next unused area within the extent or to indicate that data has been written to the EOE address. (See the diskette initialization function in the *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311, for a complete description of the index cylinder.)

Alternate Cylinders

The last two cylinders (75 and 76) are reserved for use as replacements (alternate cylinders) for defective cylinders. The remaining cylinders (1 through 74) are used for storing data.

DISKETTE TYPES AND FORMATS

The 5110 uses three types of diskettes; the one-sided diskette (1), with data recorded on just one side; the two-sided diskette (2), with data recorded on both sides; and the two-sided diskette (2D), with data recorded on both sides at double density. The diskettes are initialized (see *Disk Initialization*) into various formats consisting of:

- The number of sectors per cylinder
- The number of bytes per sector

The possible diskette formats are:

	Sectors per Track	Sectors per Cylinder	Bytes per Sector
Diskette 1	26	26	128
	15	15	256
	8	8	512
Diskette 2	26	52	128
	15	30	256
	8	16	512
Diskette 2D	26	52	256
	15	30	512
	8	16	1,024

Note: The diskette types (1, 2, or 2D) are identified on the diskette label, and the)VOLID command can be used to determine the bytes per sector (record size).

DISKETTE INITIALIZATION

The diskettes must be initialized before they can be used for storing data. All IBM-supplied diskettes are initialized before they are shipped to a customer. Reinitializing is not required unless:

- The diskette was exposed to a strong magnetic field.
- A defect occurred in one or two cylinders. In this case, initialization can be used to take the bad cylinder(s) out of service and use one or two of the alternate cylinders.
- A sector sequence other than the sequence existing on the diskette is desired.
- A format (number of sectors per cylinder) other than the existing format is desired.

See the *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311, for a description of the diskette initialization function.

VOLUME ID, OWNER ID, AND ACCESS PROTECTION

Each initialized diskette has volume ID, owner ID, and an access-protect indicator. The volume ID is the identification of the diskette volume, and the owner ID is the identification of the diskette volume owner. The access-protect indicator is used to prevent unauthorized access (reading and writing) to the diskette volume.

The)VOLID command is used to display or change the volume ID and owner ID or to change the access-protect indicator.

FILE ID

Each file header on a formatted (marked) diskette has a file ID (identification). When the diskette files are formatted, a file ID is automatically generated, even though the files are unused. For example, the file name for file 1 is SYS0001.

The)FILEID command can be used to display or change a file ID.

The file ID for a stored workspace must be a simple name. A *simple name* must begin with an alphabetic character and can be up to eight alphameric characters. For example:

SALES200

The file ID for a data file can be a simple or complex name. A *complex name* is two or more simple names with each name separated by a period. A complex name cannot exceed 17 characters including the period. For example:

SALES200.DATA

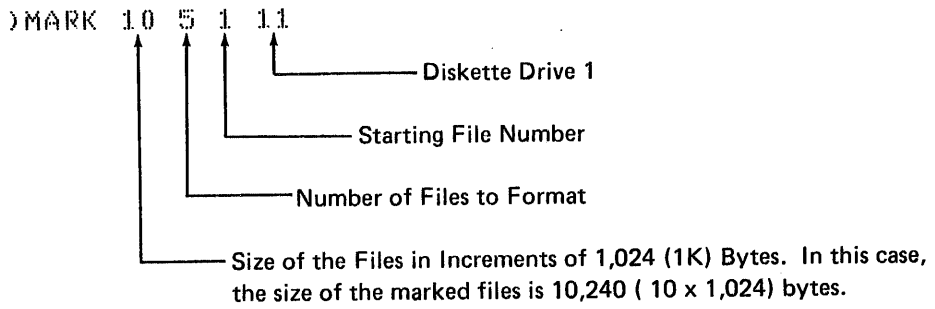
SALES.DATA.DIST12

DISKETTE FILE WRITE PROTECT

Each file header contains a write-protect indicator. When the write-protect indicator is *on*, no data can be written to the file. The)PROTECT command invokes or removes the write-protect indicator for a diskette file.

DISKETTE FILE ORGANIZATION

You use the)MARK command to format files on the diskette before you can store workspaces or data records on the diskette. For example:

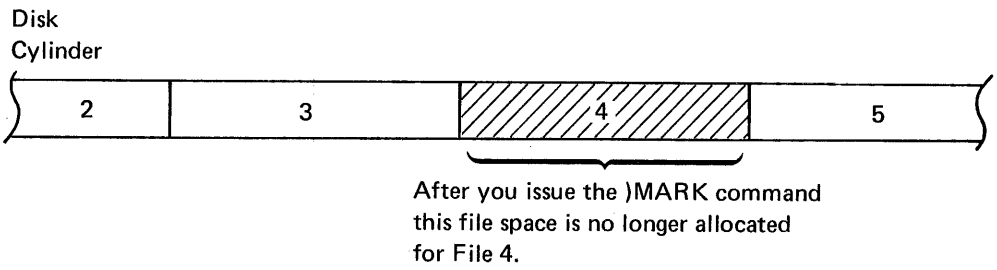


Unlike tape files, the diskette files are not always formatted sequentially on the diskette. For example, file 2 might be on cylinder 3, file 3 on cylinder 9, and file 4 on cylinder 7. You can control the location of a file on the diskette only by using a totally unmarked diskette and issuing)MARK commands in the same order as the files are to be formatted on the diskette.

When the information in a diskette file is no longer needed, you can use the)DROP command to mark the file unused so that the file is available for other uses. However, once a file is formatted, you cannot increase the size of the file without re-marking the file. Reallocating diskette file space is discussed next.

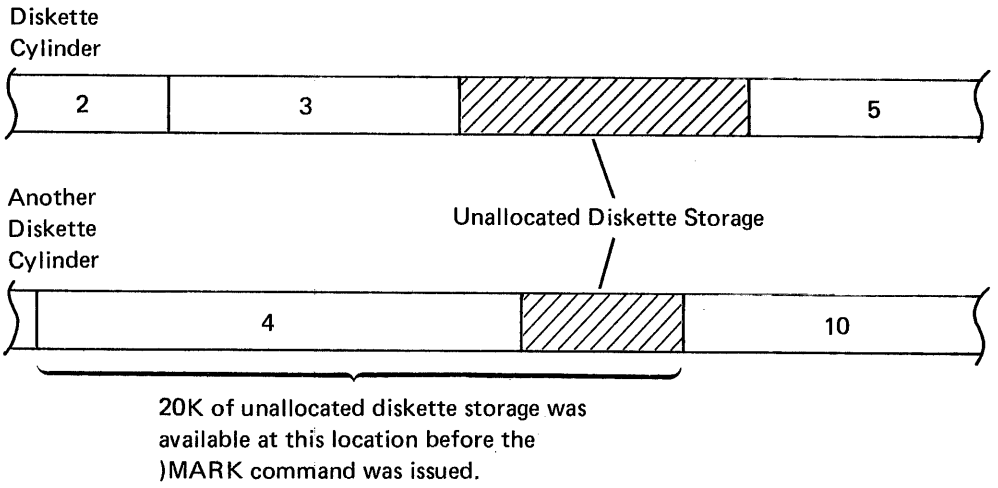
REALLOCATING DISKETTE FILE SPACE

Unlike tape files, when you re-mark an existing diskette file, no other diskette files are affected. When you re-mark a diskette file to increase the size, the file space presently allocated to that diskette file is made available for other files being marked. The remarked file will then be located on the diskette where there is enough continuous storage available for that file. For example, assume you want to increase the size of file 4 from 10K to 15K by issuing a)MARK 15 1 4 11 command:



Once the file space previously occupied by file 4 is available, that file space will be used by a subsequent)MARK command that marks a file of 10K or smaller.

After the)MARK command is successfully completed, file 4 is formatted on the diskette at a location where at least 15K of continuous storage is available.



AMOUNT OF STORAGE AVAILABLE ON A DISKETTE

The amount of storage available on a diskette depends upon:

- Whether data can be recorded on just one side or on both sides of the diskette
- The number of sectors per cylinder
- The number of bytes per sector

Each diskette has 77 cylinders. Cylinder 0 is the *index track* and is reserved for information (file headers) about the diskette files. Cylinders 75 and 76 are alternate cylinders that are used as replacements for bad cylinders. This leaves cylinders 1 through 74 available for data storage. The following chart shows the amount of storage available with the different types of diskettes:

	Sectors per Cylinder	Bytes per Sector	Available Storage in Bytes (cylinders 1-74)
Diskette 1	26	128	246,272
	15	256	284,160
	8	512	303,104
Diskette 2	52	128	492,544
	30	256	568,320
	16	512	606,208
Diskette 2D	52	256	985,088
	30	512	1,136,640
	16	1,024	1,212,416

Although the previous chart shows the maximum amount of diskette storage, the amount of diskette storage actually available to you depends on:

- The number of files and the size of the files marked on the diskette
- The types of data files that are written to the diskette
- The allocation of file space as the result of previous)MARK and)FREE commands

Number and Size of Diskette Files

The diskette volume/owner identification (7 sectors) and file header information are contained on the index cylinder. The remaining cylinders on each type of diskette can have the following maximum number of files:

	Diskette 1	Diskette 2	Diskette 2D
Maximum Number of Files	19	45	71

Note: For a type 2D diskette, see the disk initialization function in the *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311, for information on how to get additional file headers.

If you mark the maximum number of files without using all the available file space, the remaining file space becomes unavailable for storing data. For example, assume you have an unmarked Diskette 1 with 128 bytes per sector. This diskette has 246,272 bytes available for storing data; however, you issue the following command:

```
)MARK 10 19 1 1
```

- 1 — In this example, diskette drive 1 is used.
- 1 — The starting file number.
- 19 — The number of files to be marked.
- 10 — The size of each file to be marked.

This command marks the diskette with the maximum 19 files. Because each file is 10K bytes, a maximum of 190K (194,560) bytes of storage is allocated for the files. Now, if you subtract the allocated diskette storage from the available diskette file space:

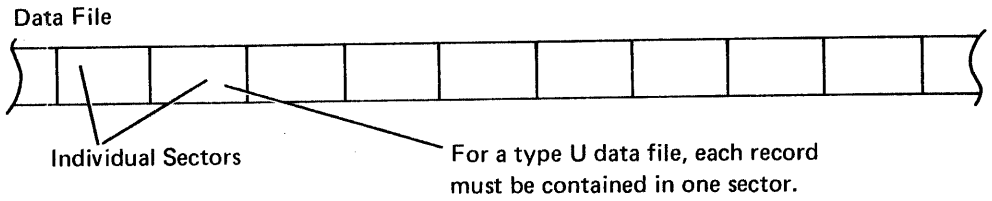
```
 246,272
-194,560
 -------
  51,712 <-- For this example, this much diskette storage is
             unavailable for you to store data.
```

Types of Data Files

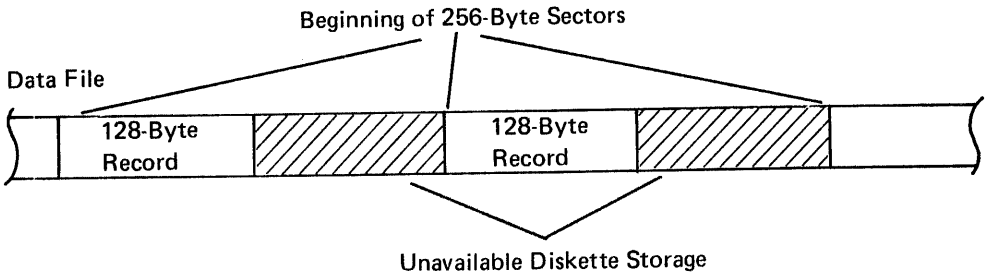
How to generate the various types of diskette data files is discussed under *Input/Output Control*. Two types of data files can cause diskette file storage to be unavailable for storing data: U (unblocked) and M (mixed).

Type U Data File

A type U data file specifies that each record in the file starts at the beginning of a sector and that a record cannot span from one sector to another.



The sectors on a diskette can be 128, 256, 512, or 1,024 bytes. If a record does not require the number of bytes available in a sector, the remaining portion of the sector is unavailable for data storage.

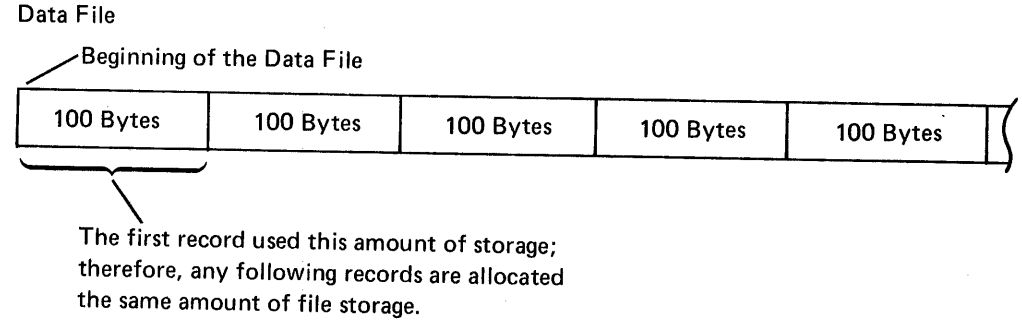


To obtain the maximum available storage for a type U data file, you should do one of the following:

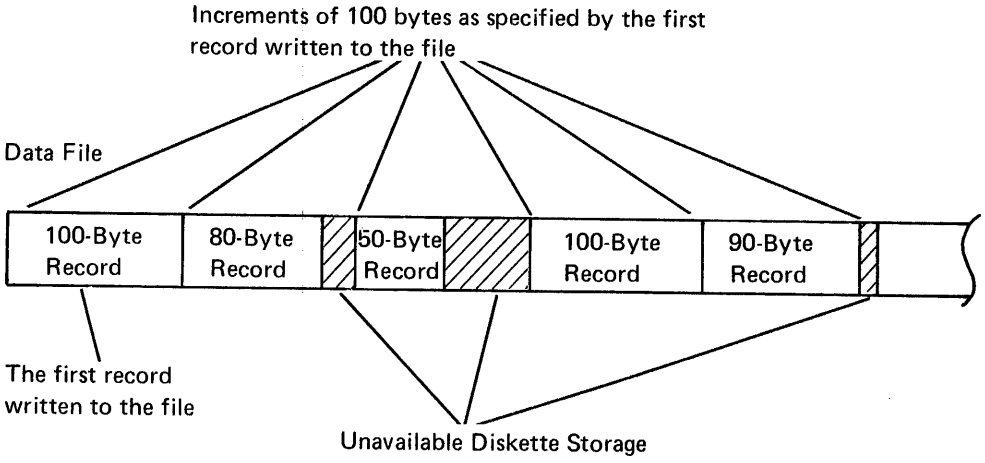
- Write records to the file that are the same size as (or as close as possible to) the sector size.
- Use a diskette whose sectors are initialized (128, 256, 512, or 1,024) nearest to the record size. Remember, the entire record must fit in one sector.

Type M Data File

A type M data file specifies that each record in the file requires the same amount of file storage as the first record written to the file. That is, the first record determines the characteristics of the file. Any record written after the first record cannot be larger than the first record.



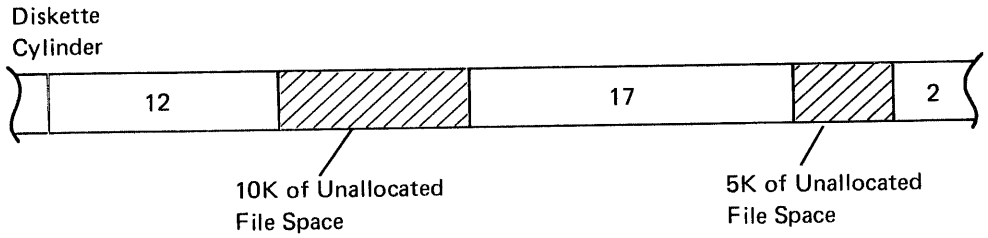
If any of the following records do not require as much storage as the first record, the remaining portion of the storage allocated for the record is unavailable for data storage.



To obtain maximum available storage for a type M data file, you should make the records as uniform in size as possible. Remember, the size of the records following the first record must be equal to or less than the size of the first record.

Allocation of File Space

Previously in this section, reallocating diskette file space using the)FREE and)MARK commands was discussed (see *Reallocating Diskette File Space*). Using the)FREE and)MARK commands to reallocate diskette file space can cause fragmented blocks of unallocated file space on the diskette. For example, assume a diskette has all the file space allocated, except the following 15K of file space on a cylinder:



Now, if you need that 15K of storage for a new file to be marked, the storage is not available because it is not in 15K continuous bytes.

The fragmented blocks of unallocated file space can be made available by the compress function (see the *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311). The compress function closes the gaps caused by the unallocated file space and places all of the unallocated file space in one continuous area.

Chapter 9. Printer Control

You can specify what data is sent to the printer by using the)OUTSEL system command or an APL shared variable. Also, at any time, you can print all the information on the display screen by holding down the CMD key and pressing the ÷ × key. When an APL shared variable is used to send data to the printer, except for using the CMD and ÷ × key, all the data to be printed must be assigned to the shared variable. (See *Input/Output Control* for more information on using shared variables for printing data.) The)OUTSEL command has three options to specify which data on the display screen is printed. These options are:

- ALL–specifies that all subsequent information displayed is printed.
- OUT–specifies that only output is printed. Even though input is displayed it is not printed. For example:

```
      5+3
8
```

If you enter this statement, the statement is displayed but not printed.

The output (result) from the statement is displayed and printed.

- OFF–specifies that none of the information displayed is printed, unless an APL shared variable is used to send the data to the printer.

You can use the following system variable and APL function to specify the format of printed output:

- ⎕PP Specifies the printing precision of numeric data (how many digits are printed)
- 5 ⎕CC n Specifies the starting print position (tab n) from the left margin
- ⍕ Formats numeric data into character data

FORMATTING OUTPUT

You can use the ⎕PP system variable to specify the number of digits to be displayed and printed for decimal numbers and for integers with more than 10 digits. In a clear workspace, the ⎕PP system variable is set to 5 (default value). For example:

```
      12345.67
12346
```

A decimal number with seven digits.

Only five digits are displayed, and the least significant digit is rounded off.

The value of ⎕PP does not affect the internal precision of the system. For example:

```
      A←12345.67
      A
12346
      ⎕PP←7
      A
12345.67
      ⎕PP←2
      A
1.2E4
```

Only five digits are printed.

Change the printing precision to seven digits.

Notice that the value of A is still seven digits.

Even though only two digits are printed, the internal value of A in the system is seven digits.

Although the ⎕PP system variable allows you to specify the number of significant digits printed for decimal numbers or integers with more than 10 digits, there might be times when even more control is needed for the printed output. The ⍕ (format) function allows you to specify the precision and spacing of numeric data. The ⍕ function also converts the numeric data to character data. This makes it easier to print the formatted numeric data with other character data. For example:

```
      'THE CLASS AVERAGE IS:',80
DOMAIN ERROR
      'THE CLASS AVERAGE IS:',80
                            ^
```

Numeric Value

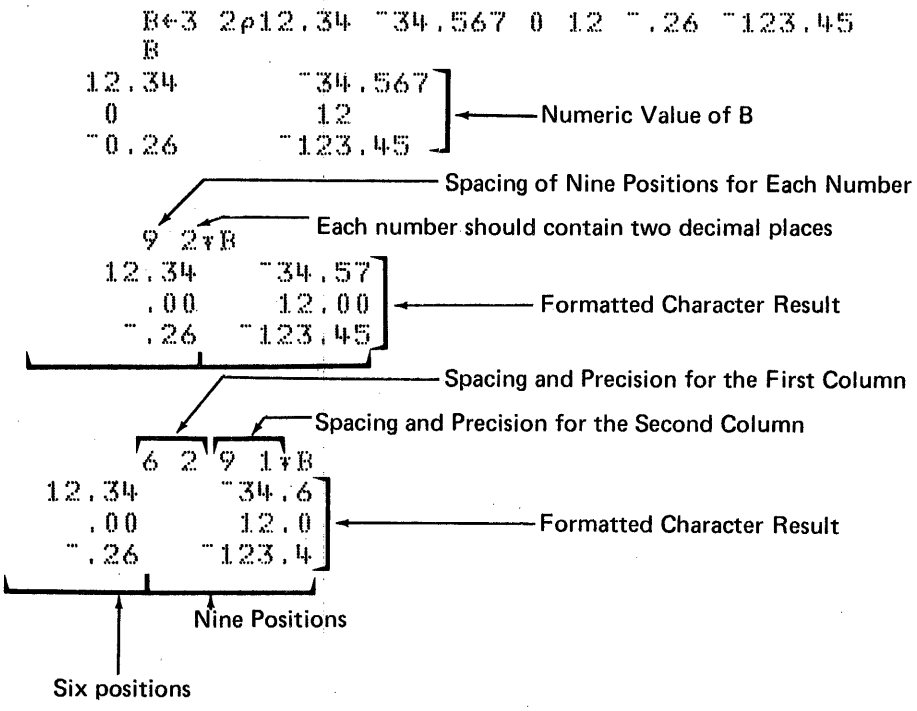
You cannot join numeric and character data

```
      'THE CLASS AVERAGE IS: ',⍕80
THE CLASS AVERAGE IS: 80
```

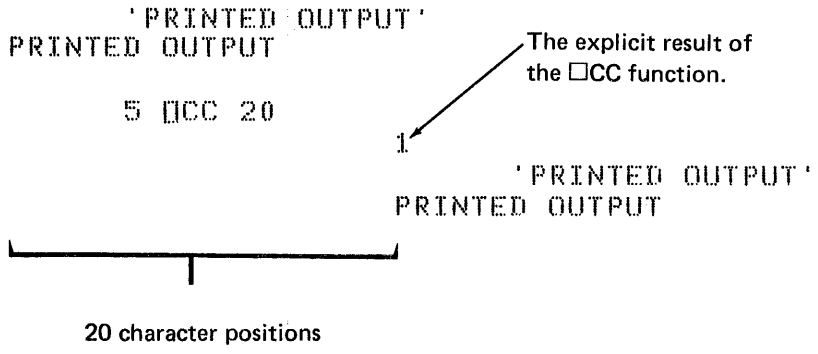
The Format Function

The Printed Result

The following examples show how the ⍕ function can be used to control spacing and precision of numeric data (see the *IBM 5110 APL Reference Manual*, SA21-9303, for a complete description of the ⍕ function):



You can use the ⎕CC (console control) system function to specify a tab position from the left margin. The printed output then starts from that tab position. To specify the tab position, the left argument of the ⎕CC function is 5 and the right argument specifies the tab position. For example:



FORMATTING REPORTS

Sometimes the data stored in the 5110 cannot be used unless the data is in a printed report. And the printed report cannot be used unless the report is in a readable format. The following procedure can be used to generate a readable report, assuming the necessary data for the report is already stored in the 5110:

1. Determine the headings for the report. The headings should describe the information in the report.

2. Arrange the data so that the data will be located under the appropriate heading.

3. Edit the data (see *Useful APL Statements and User-Defined Functions for Formatting Reports*).

4. Print the data.

The following example shows how this procedure might be used.

Assume you have the following data stored in the system:

```
      PART
392401
392402
392403
392404
392405
392406
392407
```

A numeric matrix with each row representing a part number of items in stock

```
      NAME
SCREW
NUT
WASHER
CONTACT
LEAD
POST
CRT
```

A character matrix with each row representing the name of the associated part number in matrix PART

```
 9 0 9 0 9 2 9 0⍴DATA
1000     0   .02 1000
1000     0   .01 1000
5000  2000   .01 2000
   0     0   .50 2000
  20   500  1.03  500
 400     0  2.72  500
 500   200 136.59  200
```

A numeric matrix that contains the in-stock quantity, on-order quantity, cost per part, and reorder quantity for the associated part number in matrix PART

Now, suppose you want a part inventory report that shows:

- The part number and name.
- The in-stock and on-order quantities.
- The cost per part.
- The reorder quantity and reorder flag. The reorder flag indicates that the combined in-stock and on-order quantities are less than the reorder quantity.

The first step is to determine the headings for the report. In this example, the following heading is entered:

Position 1 on Line 1

```
HEAD←2 56⍴ 'PART                              QUANTITY          COST          REO
RDER','NUMBER NAME          IN-STOCK ON-ORDER PER PART     QTY   FLAG'
```

The second step is to arrange the data so that it is located under the appropriate heading. In this example, the following statements are used to arrange the data:

```
BODY←(6 0⍕PART),'  ',NAME
BODY←BODY,8 0 9 0 12 2 8 0 ⍕DATA
```

Join the part numbers and names together.

This matrix now contains the part number, name, in-stock quantity, on-order quantity, cost per part, and reorder quantity for each part on inventory.

The spacing and precision to format each associated column of the matrix DATA.

At this point there is not enough information available to complete the report. The reorder flag is needed to indicate the parts with a combined in-stock and on-order quantity that is less than the reorder quantity. The following user-defined function might be used to place asterisks (*) in the reorder flag column when the in-stock plus on-order quantity is less than the reorder quantity:

```
      ∇REORDERFLAG[⎕]∇
    ∇ RF←REORDERFLAG;I;X
[1]   I←(DATA[;1]+DATA[;2])<DATA[;4]
[2]   X←\(1↑⍴DATA)
[3]   I←I/X
[4]   RF←((1↑⍴DATA),5)⍴' '
[5]   RF[I;]←'*'
    ∇
```

Selects the rows where the condition specified in statement 1 is true

Places the reorder flag (*****) in the appropriate rows

Now, the following statement can be used to add the reorder flag the report data:

```
BODY←BODY,REORDERFLAG
```

The body of the report now looks like this:

```
      BODY
392401 SCREW      1000        0        .02     1000
392402 NUT        1000        0        .01     1000
392403 WASHER     5000     2000        .01     2000
392404 CONTACT       0        0        .50     2000*****
392405 LEAD         20      500       1.03      500
392406 POST        400        0       2.72      500*****
392407 CRT         500      200     136.59      200
```

The third step is to edit the data. In this example, a dollar ($) is to be placed before each cost per part.

```
BODY[;3 6]←'$'
```

The fourth step is to print the report. This might be done several ways; for example:

- Join the heading and the body and then print the entire report.

```
REPORT←HEAD,[1]BODY
```

- Use a user-defined function to print out the report. For example:

```
      ∇REPORT[⎕]∇
    ∇ REPORT
[1]   HEAD
[2]   BODY←(6 0 ⍕PART),' ',NAME
[3]   BODY←BODY, 8 0 9 0 12 2 8 0 ⍕DATA
[4]   BODY←BODY,REORDERFLAG
[5]   BODY[;36]←'$'
[6]   BODY
    ∇
```

Now, when the report is printed, it looks like this:

```
      REPORT
PART                      QUANTITY        COST          REORDER
NUMBER  NAME        IN-STOCK  ON-ORDER  PER PART      QTY   FLAG
392401  SCREW           1000         0  $      .02     1000
392402  NUT             1000         0  $      .01     1000
392403  WASHER          5000      2000  $      .01     2000
392404  CONTACT            0         0  $      .50     2000*****
392405  LEAD              20       500  $    1.03       500
392406  POST             400         0  $    2.72       500*****
392407  CRT              500       200  $ 136.59        200
```

USEFUL APL STATEMENTS AND USER-DEFINED FUNCTIONS FOR FORMATTING REPORTS

Following are examples of APL statements and user-defined functions that might be useful for formatting reports.

Drop Blanks from a Character Vector:

```
      ∇DROP[⎕]∇
    ∇ Z←DROP X
[1]   Z←(X≠' ')/X
    ∇

      X←'A B C D E FG'
      DROP X
ABCDEFG
```

Drop Blanks and Periods from a Character Vector:

```
      ∇DROPBP[⎕]∇
    ∇ Z←DROPBP X
[1]   Z←(~X∈' .')/X
    ∇

      X←'A.B.C D E F'
      DROPBP X
ABCDEF
```

Replace Periods with Commas in a Character Vector:

```
      ∇REPLACE[⎕]∇
    ∇ Z←REPLACE X
[1]   X[(X='.')/⍳⍴X]←','
[2]   Z←X
    ∇

      X
A.B.C D E F
      REPLACE X
A,B,C D E F
```

Drop Leading Blanks in a Character Vector:

```
      ∇LEADING[⎕]∇
    ∇ Z←LEADING X
[1]   Z←(~∧\' '=X)/X
    ∇

      A←'     ABC  DEF'
      LEADING A
ABC  DEF
```

Left-Justify the Names in a Matrix:

```
      ∇LJ[⎕]∇
    ∇ Z←LJ X
[1]   Z←(+/∧\X=' ')⌽X
    ∇

      NAMES←3 6⍴'  SUE  JAN    DAVE  '
      NAMES
  SUE
 JAN
  DAVE
      LJ NAMES
SUE
JAN
DAVE
```

Find the Index of a Name in a Table:

```
      ∇INDEX[⎕]∇
    ∇ Z←NAME INDEX TABLE
[1]   Z←((TABLE∧.=(¯1↑⍴TABLE)↑NAME))/⍳1↑⍴TABLE
    ∇

      NAMES
SUE
JAN
DAVE
      'JAN' INDEX NAMES
2
```

Insert Blank Lines into a Character Matrix:

```
      ∇BLANKLINES[⎕]∇
    ∇ Z←X BLANKLINES M
[1]   Z←(((1↑⍴M)+⌊(1↑⍴M)÷X)⍴(X⍴1),0)⍀M
    ∇

      NAMES←6 4⍴'DON DAN SUE JIM TOM KIM '
      NAMES
DON
DAN
SUE
JIM
TOM
KIM
      2 BLANKLINES NAMES
DON
DAN

SUE
JIM

TOM
KIM
```

The left argument determines how often the blank lines are inserted.

Print a Matrix Using a Shared Variable:

```
      ∇PRINT[⎕]∇
    ∇ PRINT X;Z;P
[1]   CR←⎕AV[⎕IO+156]
[2]   P←'PRT'
[3]   Z←1 ⎕SVO 'P'
[4]   P←¯1↓,(⍕X),CR
    ∇

      NAMES←3 4⍴'SUE JAN DAVE'
      NAMES
SUE
JAN
DAVE
      PRINT NAMES
SUE
JAN
DAVE
```

Drop Leading Blanks or More than One Consecutive Blank from a Character String:

```
      ∇SCAN[⎕]∇
    ∇ Z←SCAN X
[1]   Z←1↓((X≠' ')∨' '≠1⌽X)/X←' ',X
    ∇

      A←'   A  B  C    D EF'
      SCAN A
A B C D EF
```

Convert a Scalar or Vector to a Matrix:

```
      ∇CONVERT[⎕]∇
    ∇ Z←CONVERT X
[1]   →(0 1 2 =ρρX)/(SCALAR,VECTOR,MATRIX)
[2]  MATRIX:Z←X
[3]   →0
[4]  SCALAR:Z← 1 1 ρX
[5]   →0
[6]  VECTOR:Z←(1,ρX)ρX
    ∇

      A←1
      ρA

      A←CONVERT A
      ρA
1 1
```

Join Two Variables Together Side by Side:

```
      ∇BY[⎕]∇
    ∇ Z←A BY B;ROW
[1]   A←(2↑(ρA), 1 1)ρA
[2]   B←(2↑(ρB), 1 1)ρB
[3]   ROW←1↑(ρA)⌈ρB
[4]   A←(ROW,¯1↑ρA)↑A
[5]   B←(ROW,¯1↑ρB)↑B
[6]   Z←A,B
    ∇

      A←2 2ρ'A'
      B←3 3ρ'B'
      A BY B
AABBB
AABBB
  BBB
```

Join Two Variables Together One on Top of the Other:

```
      ∇ON[⎕]∇
    ∇ Z←A ON B;COL
[1]   A←(¯2↑ 1 1 ,⍴A)⍴A
[2]   B←(¯2↑ 1 1 ,⍴B)⍴B
[3]   COL←¯1↑(⍴A)⌈⍴B
[4]   A←((1↑⍴A),COL)↑A
[5]   B←((1↑⍴B),COL)↑B
[6]   Z←A,[1] B
    ∇

      A←2 2⍴'A'
      B←3 3⍴'B'
      A ON B
AA
AA
BBB
BBB
BBB
```

Chapter 10. Input/Output Control

Input/output operations consist of the following:

- Reading and writing data records on a tape or diskette data file
- Reading and writing data on the display screen
- Printing data records

In this chapter, the following topics concerning input/output control are discussed:

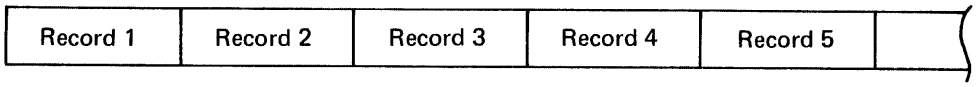
- Sequential and direct access data files
- Logical and physical records
- Types of data file formats
- APL internal code to EBCDIC (extended binary coded decimal interchange code) translation
- The 5110 I/O processor
- Establishing an APL shared variable as the connection between the active workspace and the I/O processor
- Using the APL shared variable for input/output operations
- Retracting the shared variable offer

SEQUENTIAL AND DIRECT ACCESS DATA FILES

Sequential access data files can be on tape or diskette: however, direct access data files can only be on diskette. The data file is specified as either a sequential access or direct access data file when it is initially created (see *Specifying the Operation to be Performed* later in this chapter).

All 5110 data files are created sequentially: that is, individual records are written to the file in sequential order.

Data File



After a data file is created, data can be read from the file as follows:

- For a sequential access file, the records are read in the same sequence as the records were written to the file. For example, the first and second record must be read before the third record can be read. Also, if you want to read a record previous to the last record read, you must start the operation over from the beginning of the data file.

- For a direct access data file, you specify the record(s) that are to be read from the data file. For example, you can specify that the fourth record in the file be read (without reading the previous three records) then after the fourth record is read you can specify that the second record in the file be read, and so on. Also, multiple records can be read with one statement.

LOGICAL AND PHYSICAL RECORDS

A logical record is the individual record written to a data file. For sequential access data files, the size of each logical record can vary. For example, one logical record might require 10 bytes of storage and the next logical record might require 20 bytes of storage. For direct access data files, each logical record requires the same amount of diskette storage. Generally, each logical record must be the same shape and representation as the other logical records in the data file.

A physical record is a certain amount of tape or diskette storage. For tape, a physical record is 512 bytes. For diskette, a physical record is the same as the diskette sector size (128, 256, 512, or 1024 bytes).

TYPES OF DATA FILE FORMATS

For Sequential Access Data Files

There are two types of data file formats for sequential access data files:

- APL internal format
- General exchange format

Following is a description of these data file formats.

APL Internal Format

The APL internal format allows data to be written to the data file in the same format as that stored in the active workspace. For example, the first record written to the data file might be a numeric 10 x 20 matrix, and the second record written to the data file might be a 10-element character vector.

General Exchange Format

The general exchange data format is the basis of exchange between the 5110 APL and BASIC languages. The general exchange format only allows character scalars or vectors to be written to the data file. Therefore, when the system is storing numeric data in data files with the general exchange data format, the format function must first be used to change the data to a character scalar or vector.

The following rules apply to a general exchange format data file that is written using the APL language for later processing using the BASIC language.

1. All data items must be separated by commas. For example, the numeric vector 1 3 5 6 must be changed to character data, then commas must be placed in the blank positions. The following statement replaces blanks with commas in a character vector:

   ```
   X[(X=' ')/⍳⍴X]←','
   ```
   ← where X is the character vector.

2. Negative signs must be replaced by minus signs.

3. The 5110 BASIC language accepts only the first 255 characters in each character constant.

4. The 5110 BASIC language creates a logical record for each PUT statement or each row of an array with a MAT PUT statement.

When the 5110 APL language is used to read a general exchange format data file (see *To Read a Sequential Access Data File* later in this chapter), the following actions are taken by the 5110 if a cursor return character (hex 9C) or end-of-block character (hex FF) was embedded in a character vector that was written to the data file:

- If a cursor return character was embedded in the character vector, the data will be read from tape in a different sequence than it was written to tape. This condition occurs because as the interchange data is written to tape, the system writes an end-of-record character (hex 9C) after each character vector (record) that was written to tape. The end-of-record character and the cursor return character are the same. When used on tape, this character separates the data (records) so that it can be read from tape in the same sequence as it was written to tape. However, if a cursor return character is embedded in the data that was written to tape, the system will recognize it as an end-of-record character when the data is read from tape.

- If an end-of-block character was embedded in the character vector, any data from the embedded end-of-block character to the next physical record is not read from tape. This condition occurs because the system looks at the tape in 512-byte segments (one physical record). A physical record can be terminated by an end-of-block character (hex FF). When the system is reading data from the tape and an end-of-block character is encountered, the system skips to the next physical record and continues reading data. Therefore, if an hex FF character is embedded in the data that was written to tape, the system recognizes it as an end-of-block character when the data is read from tape and skips ahead to the next physical record.

For Direct Access Data Files

There are five types of data file formats for direct access data files. These formats are:

- APL internal format
- General exchange format
- Unblocked and unspanned format
- Mixed format
- Nontranslated format

Following is a description of these data file formats.

APL Internal Format

The APL internal format allows data to be written to the data file in the same format as that stored in the active workspace. However, unlike sequential access data files, all the records written to the data file must have the same shape and internal representation. For example, all of the records written to the data file must be one of the following:

- Character data
- Numeric binary data (all zeros and ones)
- Numeric fixed point (all integers in the range -2^{31} to $2^{31}-1$)
- Numeric floating point (all other values)

These records must have the same shape and representation because, when multiple records are read from a direct access data file, the records are laminated (joined) together along a new first dimension (see *Updating a Direct Access Data File* later in this chapter).

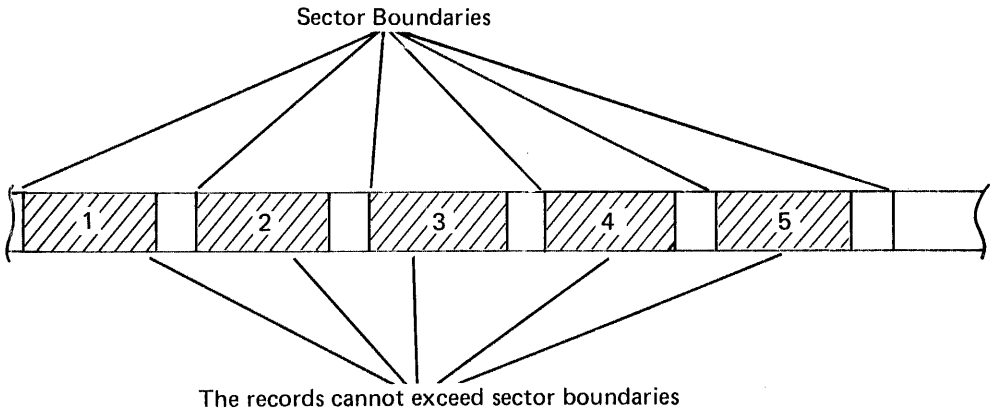
General Exchange Format

The general exchange format for direct access data files is similar to the general exchange format for sequential access data files (see *For Sequential Access Data Files* under *Types of Data File Formats* in this chapter). However, when the general exchange format is used with direct access data files, all the records written to the data file must be character data and must have the same shape (⍴ DATA) as the first record written to the data file. The records must have the same shape because, when multiple records are read from a direct access data file, the records are laminated together along a new first dimension.

Unblocked and Unspanned Format

The unblocked and unspanned format is the basis for general exchange with other products using diskette storage. In this case, the file name cannot exceed 8 alphameric characters, and the diskette sector size must be 128 or 256 bytes.

The unblocked unspanned format allows only character scalars or vectors to be written to the data file, with each record starting on a sector boundary. Also, the records cannot exceed (span) a sector boundary.

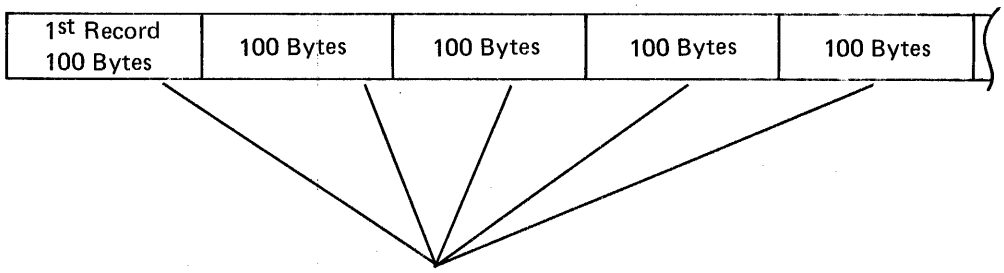


When the unblocked and unspanned format is used, all the records written to the data file must have the same shape (ρ DATA) as the first record written to the data file. The records must have the same shape because, when multiple records are read from a direct access data file, the records are laminated together along a new first dimension.

Since the unblocked unspanned format requires each record to start on a sector boundary, to prevent wasted diskette storage the record size should be as close as possible to the sector size. The unused diskette storage in each sector is unavailable for storing data (see *Type U Data File* in Chapter 8, *Diskette Concepts*).

Mixed Format

The mixed format, like the APL internal format, allows data to be written to the data file in the same format as the data is stored in the active workspace. However, the mixed format also allows you to write records that have different internal representations on the same data file. For example, you can write character data, logical data, numeric fixed-point data, and numeric floating-point data on the same data file. With the mixed format, the first record written to the data file determines how much file storage is allocated for each additional record to be written to the data file. For example:



If the first record requires 100 bytes of file storage, 100 bytes of storage are allocated for each additional record. See *Storage Requirements* in the *5110 APL Reference Manual* for information on how many bytes of storage are required for each data type.

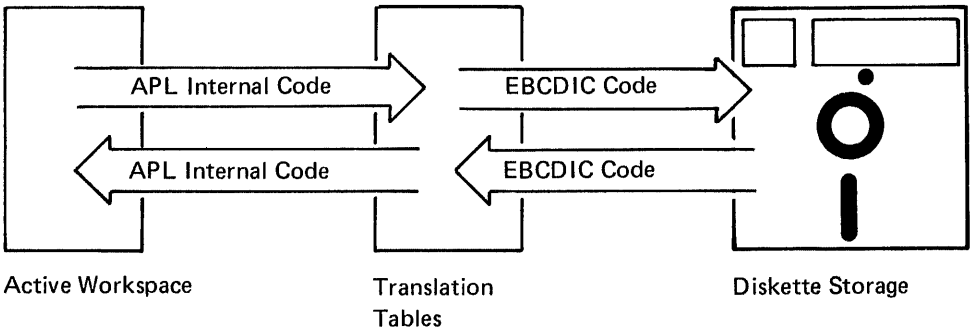
As various size records are written to a mixed format data file, any unused diskette storage allocated for each record is unavailable for storing data (see *Type M Data File* in Chapter 8, *Diskette Concepts*).

Because the records in a mixed format data file do not have to be the same shape or internal representation, only one record at a time can be directly accessed from the data file. That is, multiple records cannot be laminated together.

Nontranslated Format

The nontranslated format is used primarily for reading diskettes from a non-5110 system and the diskette does not have standard 5110 file types. Therefore, unless you need to read diskettes that do not have standard 5110 file types, you might want to skip this topic.

The nontranslated format is the only format that is not specified when the data file is initially created. The nontranslated format can be specified only when the system is reading from or writing to any diskette data file that already exists. When data is written to a direct access data file using the general exchange or unblocked unspanned format, the APL internal code is translated to EBCDIC code before the data is written on the diskette. This EBCDIC code is then translated back into the APL internal code when the data is read back into the active workspace.



However, when the nontranslate format is specified, all attributes of the file are ignored and *any* existing diskette file can be read from or written to. In this case, as the data is being read from or written to the file, the data is not translated from the EBCDIC or APL internal code to the other code.

Instead, as data is being read in the nontranslate format from an EBCDIC diskette file, the EBCDIC representation of the data is placed in the active workspace. Or when data is being written in the nontranslate format to the diskette file, the APL internal representation of the data is written to the diskette file.

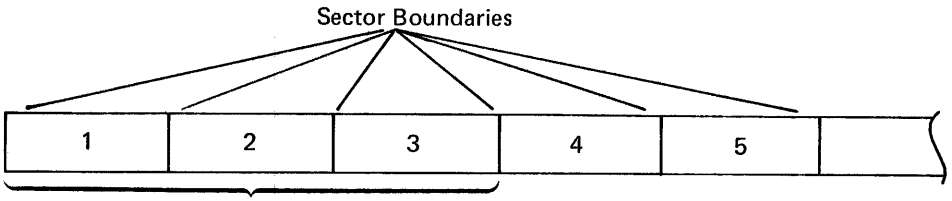
Note: Assuming ⎕IO is 0, the APL internal representation of a character is equivalent to the hexadecimal value of that character's index in the atomic vector. For example, the APL internal representation of the character A (⎕AV [86]) is hex 56 (the bit value is 01010110). You can determine the bit value of the atomic vector index using the encode (⊤) function. For example:

```
      2 2 2 2 2 2 2 2 ⊤ 86
0 1 0 1 0 1 1 0
```

Hexadecimal 56

The EBCDIC character A has a hexadecimal value of C1 or a decimal value of 193. In non-translate mode (and ⎕IO←1), when the EBCDIC character A is read from a diskette, the character is stored internally and displayed as a lowercase h (⎕AV [193]).

The nontranslate format allows only character vectors to be written to the file. Also, each sector on the diskette is considered one record. Therefore, if you specify that three records should be read or written, you will read or write three sectors on the file.



When nontranslate format is used,
three records are equal to three sectors.

THE 5110 I/O PROCESSOR AND SHARED VARIABLES

The 5110 I/O processor is responsible for transferring data from the active workspace to the tape, the diskette, the display screen, or the printer, and for transferring data from the tape, the diskette, or the display screen to the active workspace. Before individual data records can be transferred by the I/O processor, a shared variable must be established as the connection between the active workspace and the I/O processor. That is, the variable is shared between the active workspace and the I/O processor.

Once this connection is established, the shared variable is used to send control information and data to the I/O processor and to receive return codes and data from the I/O processor.

ESTABLISHING AN APL SHARED VARIABLE

A variable must be offered to the I/O processor before the variable can become a shared variable. To do this, you must use the ⎕SVO (shared variable offer) system function. The ⎕SVO function requires two arguments; the left argument must be 1 (to specify the 5110 I/O processor), and the right argument must be character data that represents the variable name(s) to be shared. If more than one name is required, the names may be entered as a character matrix with each row representing an individual name. For example:

```
1 ⎕SVO 'A'
```

The variable name A is offered to be shared.

```
1 ⎕SVO 3 1⍴'ABC'
```

The variable names A, B, and C are offered to be shared.

The ⎕SVO function generates an explicit result of 2 for each variable name that is successfully established as a shared variable with the I/O processor. For example:

```
      X←1 ⎕SVO 3 1⍴'ABC'
      X
2 2 2
```

There are *two* users of the shared variable, the active workspace and the I/O processor. This is called the degree of coupling.

A 0 or 1 is the result of the ⎕SVO function for each variable name that is not successfully shared with the I/O processor. If the result is 1, the left argument of the ⎕SVO function was a value other than 1. In this case, the variable name must be retracted and offered as a shared variable again with a 1 as the left argument of the ⎕SVO function (retracting the shared variable name is discussed later in this chapter). If the result is 0, an error message is also displayed.

You can establish up to 12 shared variables in the 5110 active workspace. If you attempt to establish more than 12 variable names as shared variables, the error message INTERFACE QUOTA EXHAUSTED is displayed. The statement `(0≠⎕SVO ⎕NL 2)/⎕NL 2` displays the existing shared variable names in the active workspace. If 12 shared variable names are already established, you must retract a shared variable name before another variable name can be offered.

USING APL SHARED VARIABLES

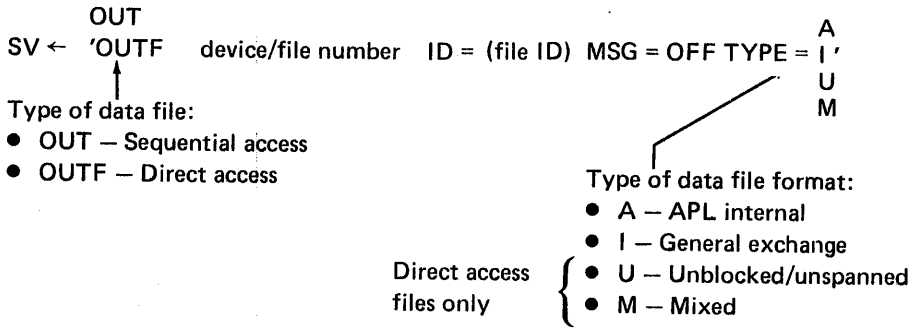
Once a shared variable is established as the connection between the active workspace and the I/O processor, you can do the following input and output operations:

- Create a new sequential access or direct access data file.
- Add data to an existing data file.
- Read data from a sequential access data file.
- Read-only data from a direct access data file.
- Update (read and write) data in a direct access data file.
- Read data from and write data to the display screen.
- Send data to the printer.

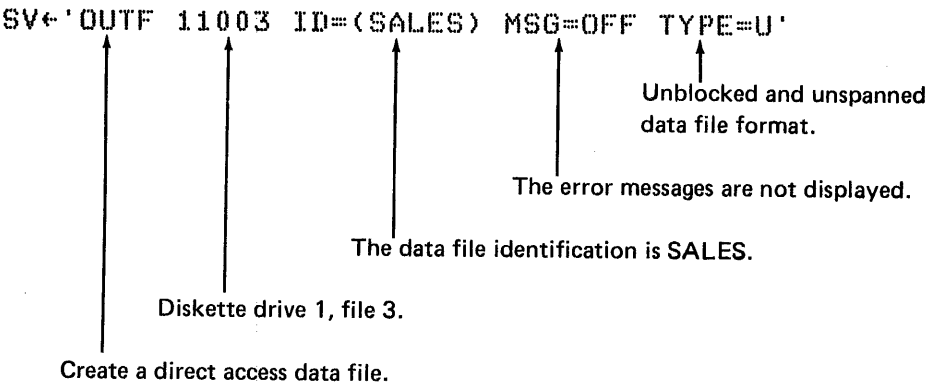
The first value assigned to the shared variable must be a character string (enclosed in single quotes) that specifies the operation to be performed. Specifying the operations to be performed and doing input/output operations are discussed next.

To Create a New Sequential Access or Direct Access Data File

The first value assigned to the shared variable must specify the operation to be performed as follows:



For example:



(See the *IBM 5110 APL Reference Manual* for a detailed description of each parameter.)

Once the operation to be performed is specified to the I/O processor, the I/O processor assigns a return code to the shared variable. For example:

```
      SV←'OUTF 11003 ID=(SALES) MSG=OFF TYPE=U'
      SV
0 0
```

The return code is checked by referencing the shared variable.

The return code is a two-element vector.

In the previous example, the return code is 0 0, which indicates that the operation to be performed is successfully specified to the I/O processor. See the *IBM 5110 APL Reference Manual* for a description of all the return codes.

Once the operation to be performed is successfully specified, the shared variable used to specify the operation is then used to specify data that is to be written on the data file. That is, each time new data is assigned to the shared variable, the I/O processor transfers that data to a data file and assigns a return code to the shared variable. For example:

```
      SV←12↑'RECORD ONE'
      SV
0 0
      SV←12↑'RECORD TWO'
      SV
0 0
      SV
IO STATUS: INVALID OPERATION
INTERRUPT
      SV
      ^
      SV←12↑'RECORD THREE'
      X←SV
      X
0 0
      X
0 0
```

Remember, in this example, the data file format is unblocked and unspanned. All of the records in the data file must have the same shape.

Notice that the shared variable cannot be referenced more than once to check the return code.

The return code can be assigned to another variable and then checked more than once.

When all of the data is written to the data file, the operation must be terminated. The operation can be terminated by assigning an empty vector to the shared variable. For example:

```
      SV←⍳0 ←———————— Empty vector
      SV
0 0 ←———————— The operation is terminated. The variable
              SV is still shared with the I/O processor.
              Therefore, SV can be used to specify and
              perform another input/output operation.
```

See the *IBM 5110 APL Reference Manual* for a description of other ways to terminate the operation.

To Add Data to an Existing Data File

There will be times when you want to add data to an existing data file. If you specify an OUT or OUTF operation, the data you write to the file always starts at the beginning of the file and the new data is written over any existing data. Therefore, to add data to a data file starting after the last record in the file, you must specify the operation to be performed as follows:

SV ← 'ADD [device/file number] [ID = (file ID)] [MSG = OFF]'

For example:

```
      SV←'ADD 11003 ID=(SALES) MSG=OFF'
      SV
0 0
```

Do not display the error messages.

The file identification of the existing data file.

The data file is on diskette drive1, file 3.

Add data starting after the last record in the data file.

Even though the device/file number and ID = (file ID) parameters are optional, one or the other of these parameters must always be specified.

The type of data file (sequential access or direct access) and data file format are determined by the attributes of the existing data file. In this case, assume this example is continued from the previous topic, the file is a direct access, unblocked, and unspanned data file.

Once operation is specified, data can be sequentially written to the data file. For example:

```
      SV←12↑'RECORD FOUR'
      SV
0 0
      SV←12↑'RECORD FIVE'
      SV
0 0
      SV←⍳0
      SV
0 0
```

You can terminate the operation by assigning an empty vector to the shared variable.

To Read a Sequential Access Data File

Once a sequential access file is created, the records are sequentially read from the data file. That is, the records are read from the data file in the same sequence as the records were written to the data file. You specify the operation to be performed as follows:

SV← 'IN [device/file number] [ID = (file ID)] [MSG = OFF]'

For example:

```
      SV←'IN 11004 ID=(SEQ) MSG=OFF'
      SV
0 0
```

Do not display the error messages.

The existing data file identification.

The data file is on diskette drive 1, file 4.

Read data from a sequential access data file.

After the operation is specified, the I/O processor assigns a return code to the shared variable. The I/O processor then assigns a new record from the data file to the shared variable each time the shared variable is referenced. For example: First, to create a sequential access file with 5 records.

```
      1 ⎕SVO 'SV'
2

      ∇SEQUENTIAL[⎕]∇
    ∇ SEQUENTIAL
[1]   SV←'OUT 11004 ID=(SEQ) TYPE=A'
[2]   I←1
[3]  LOOP:→(∨/0≠X←SV)/ERROR
[4]   SV←'RECORD ',⍕I
[5]   →(5≥I←I+1)/LOOP
[6]   SV←⍳0
[7]   →(∧/0=X←SV)/0
[8]  ERROR:'CREATING THE DATA FILE FAILED. THE RETURN CODE IS:
       ',⍕X
    ∇

      SEQUENTIAL
```

Now, to read the 5 records from the data file.

```
      SV←'IN ID=(SEQ)'
      SV
0 0
      SV
RECORD 1
      SV
RECORD 2
      SV
RECORD 3
      X←SV
      X
RECORD 4
      SV
RECORD 5
      SV

      SV
0 0
```

You should check the return code after the operation is specified.

After the return code is checked, each time the shared variable is referenced, the I/O processor assigns the next record from the data file to the shared variable.

If you want to save a record for later use, you must assign the shared variable to another variable.

After the last record is read from the file, an empty vector (s0) is assigned to the shared variable. This empty vector terminates the operation. (In a user-defined function, you can check for the empty vector. For example: → (0 = ρX ← SV)/DONE. The statement branches to DONE if the last record read was an empty vector.)

When reading a sequential access data file, the I/O processor does not assign return codes to the shared variable. However, if an error occurs, an empty vector is assigned to the shared variable and the operation is terminated. Then the I/O processor assigns a return code to the shared variable that indicates why the error occurred.

You can also terminate the operation at any time by assigning an empty vector to the shared variable. For example:

```
      SV←'IN ID=(SEQ)'
      SV
0 0
      SV
RECORD 1
      SV
RECORD 2
      SV←⍬
      SV
0 0
```

The read operation is terminated.

To Update Data in a Direct Access Data File

Once a direct access data file is created, specified records in the data file can be updated. That is, records can be read from the data file, updated in the active workspace, and then written back to the data file at a specified record location. Unlike the operations discussed so far, reading or writing specified records on a direct access data file requires a pair of shared variables. One of the shared variables is used to specify control information to the I/O processor. The I/O processor also assigns the return codes to this shared variable. The other shared variable is used for data that is written to or received from the data file. The pair of shared variable names must have the following characteristics:

- The shared variable name used to specify the control information must have the 3-character prefix CTL.
- The shared variable name used for the data must have the 3-character prefix DAT.
- After the 3-character prefix, the next 15 characters in each name must be identical, if specified. For example:

```
CTL ]
DAT ]
 or
CTL NAME ]
DAT NAME ]
```

These variable names can be used as a pair of shared variables.

Once the pair of shared variables is established, you specify the operation to be performed as follows:

```
CTL ← 'IOR [device/file number] [ID = (file ID)] [MSG = OFF] [TYPE = N]'
      IORH
```

Read or write to a direct access data file. See the *IBM 5110 APL Reference Manual* for a description of the differences between an IOR and IORH operation.

If this parameter is specified, the data is not translated (see *Nontranslated Format*). The other formats cannot be specified when you are reading from or writing to a direct access data file, because these formats were established when the file was created.

For example:

```
      CTL←'IOR 11005 ID=(DIRECT) MSG=OFF'
      CTL
0 x
```

Do not display the error messages.

The file identification.

The data is on diskette drive 1, file 5.

Read records from or write records to a direct access data file.

For a read or write operation to a direct access data file, the second element of the return code identifies the number of records in the data file.

After the operation is specified, the CTL shared variable is then used to specify what records to read or write. To do this, you must assign a two- or three-element vector to the CTL shared variable, as follows:

```
          0
CTL ← 1     starting record number [number of records]
          2
```

0 specifies a *read* records operation.
1 specifies a *write* records operation.
2 specifies a *search by key* operation. (search by key is discussed later in this section).

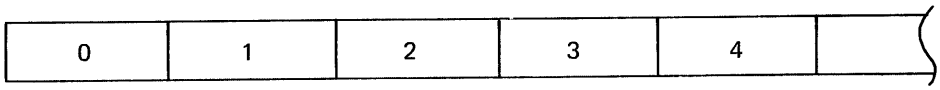
If the third element is not specified, one record is assumed.

For example:

```
CTL←0 5 3
```

Read three consecutive records, starting with the sixth record in the file.

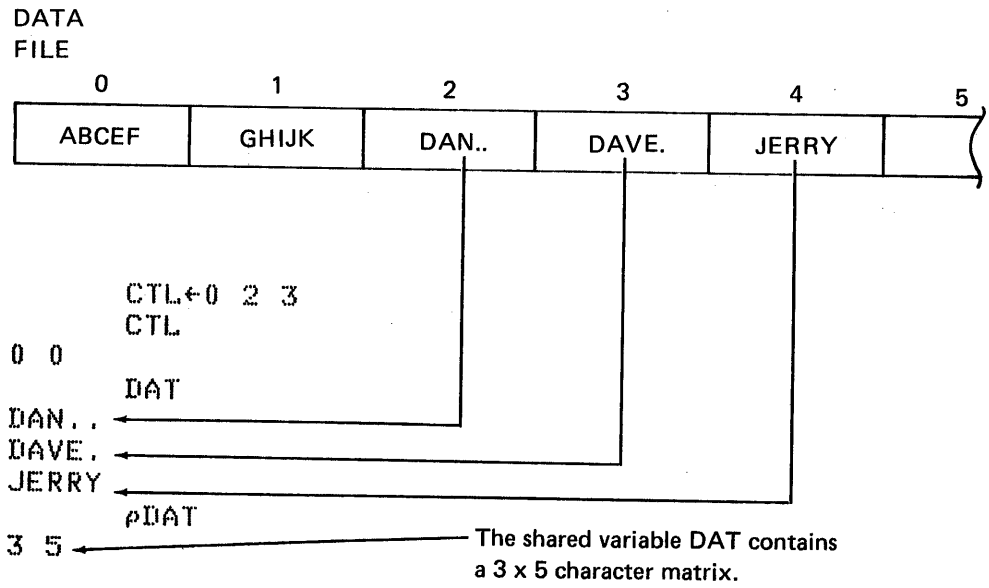
Note: The record numbering starts with zero.



Therefore, the first record in the file is record number 0, the second is record number 1, and so on. Also, you cannot read from or write to a record position that was not originally created using an OUTF or ADD operation.

When the CTL shared variable specifies a read records operation, the I/O processor assigns the records read from the data file to the DAT shared variable. When the CTL shared variable specifies a write records operation, the data currently assigned to the DAT shared variable is written to the data file. Therefore, the data must be assigned to the DAT shared variable before you specify the write operation. In each case, after reading or writing records, the I/O processor assigns a return code to the CTL shared variable.

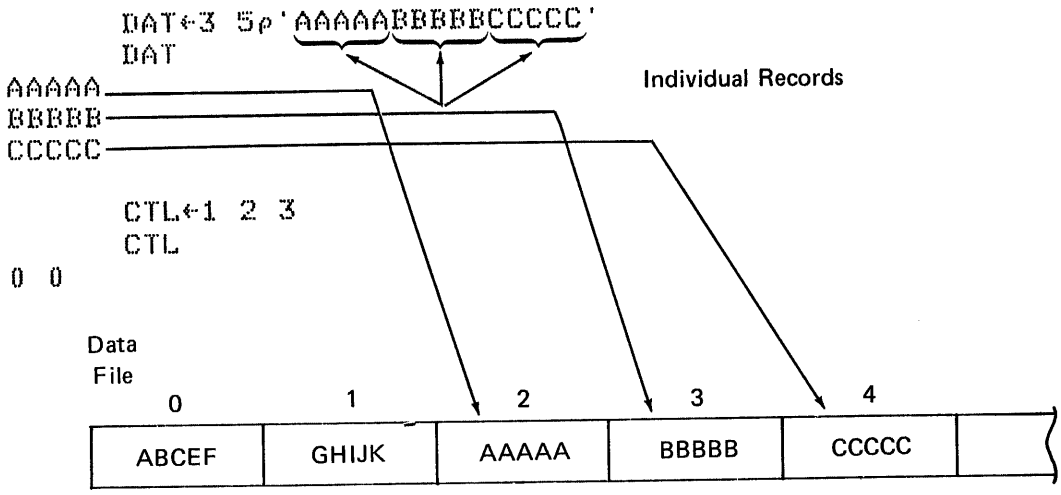
When you read multiple records from a direct access data file, the records are laminated together along a new first dimension before they are assigned to the DAT shared variable. For example, assume you read three records (character vectors) from the following data file:



The new first dimension represents the number of records read. You can specify each individual record by indexing the new dimension.

```
      DAT[1;]
DAN..
      DAT[3;]
JERRY
```

When you write multiple records to a direct access data file, the records must be joined along a new first dimension before they are assigned to the DAT shared variable. For example, assume you want to write the following three records (character vectors) to a direct access data file:



Following is an example of updating records in a direct access data file:

```
      1 ⎕SVO 2 4⍴'CTLXDATX'
2 2
```

Establish a pair of shared variables.

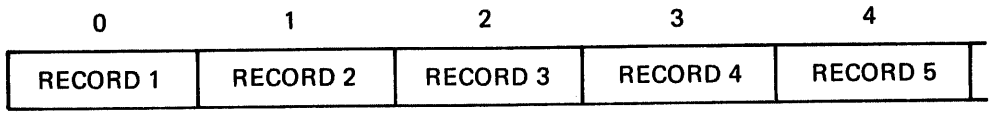
```
      ∇DIRECT[⎕]∇
    ∇ DIRECT;I;X
[1]   CTLX←'OUTF 11004 ID=(DIRECT) TYPE=I'
[2]   →(0≠1↑X←CTLX)/ERROR
[3]   I←1
[4]  LOOP:CTLX←'RECORD ',⍕I
[5]   →(0≠1↑X←CTLX)/ERROR
[6]   →(5≥I←I+1)/LOOP
[7]   CTLX←⍳0
[8]   →(0≠1↑X←CTLX)/ERROR
[9]   →0
[10] ERROR:'CREATING THE DATA FILE FAILED.'
[11]  'THE RETURN CODE IS: ',⍕X
    ∇
      DIRECT
```

Create a direct access data file.

CTLA and DATA can be used for any input/output operation.

This operation is terminated; now CTLA can be used for other operations.

The data file now looks like this:



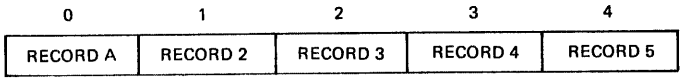

```
      CTLX←'IOR 11004 ID=(DIRECT)'
      CTLX
0 5
```

When you specify the operation to a direct access data file, the second element of the return code is the number of records in the file.

```
      CTLX←0 0
      CTLX
0 0
      DATX
RECORD 1
```

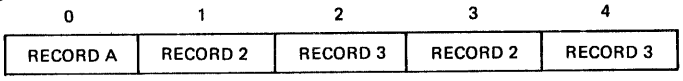
Data records can be read, updated, and written back at the same record location.

```
      DATX←'RECORD A'
      CTLX←1 0
      CTLX
0 0
```



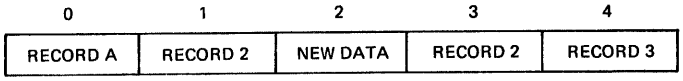

```
      CTLX←0 1 2
      CTLX
0 0
      DATX
RECORD 2
RECORD 3
      CTLX←1 3 2
      CTLX
0 0
```

Data records can be moved from one record location to another record location.




```
      DATX←'NEW DATA'
      CTLX←1 2
      CTLX
0 0
```

New records can replace existing records in the data file.




```
      CTLX←⍬0
      CTLX
0 0
```

Remember, when you write records to the data file, the records must meet the requirements of the data file.

Terminate the operation. Now CTLX and DATX can be used for other input/output operations.

To Search by Key a Direct Access Data File

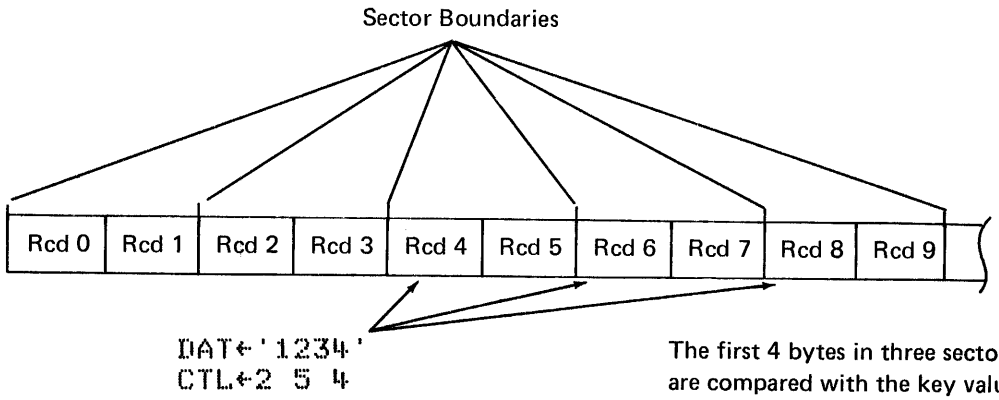
When doing update operations to a direct access data file, you can search the file for a specific record by first assigning a key value to the DAT shared variable and then assigning the following vector to the CTL shared variable:

```
CTL ← 2    starting record number of records
```

Specifies a search by key.

Unlike reading from and writing to a direct access data file, the third element must be specified for a search by key.

When a search by key is specified, the key value currently assigned to DAT is checked against the equivalent number of beginning bytes in the specified sectors. The specified sectors include the sector that contains the starting record number through the sector that contains the last record specified (determined by the number of records specified). For example:



When you are creating a file for search by key operations, the file should meet the following requirements:

- The records are sorted in ascending sequence.
- The records do not span sector boundaries. However, there can be multiple records in a sector.
- A record that is greater than any key value that might be specified should start on the sector boundary following the last valid data record.

See the *IBM 5110 APL Reference Manual* for a description of when a search by key is complete. If the search is successful, the record(s) in the sector containing the appropriate record is assigned to the DAT shared variable (multiple records are laminated together along a new first dimension). Also, the second element of the return code assigned to the CTL shared variable represents the record number of the first record assigned to the DAT shared variable.

```
      CTL
0 4
```

The search was successful, and the first record assigned to DAT is record number 4.

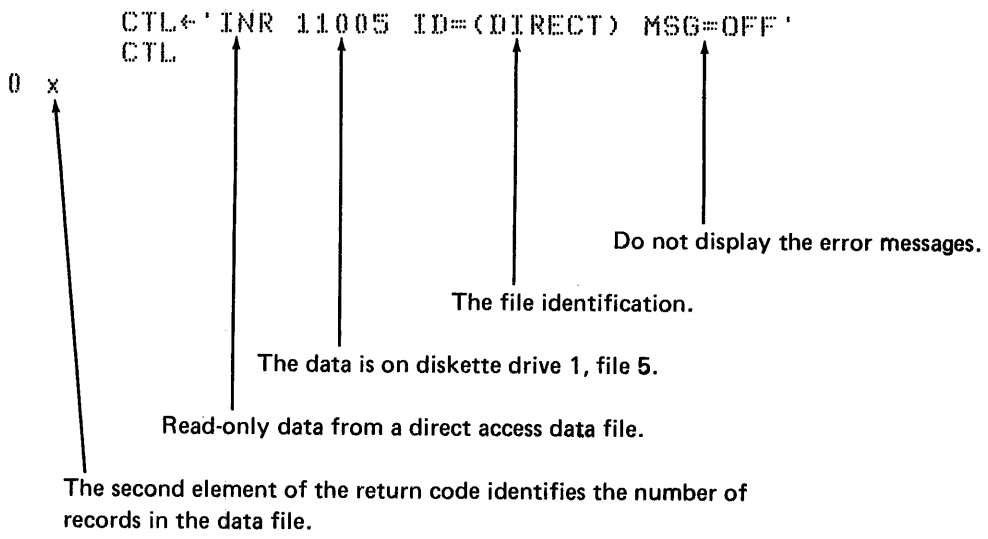
To Read-Only Data from a Direct Access Data File

To read-only data from a direct access data file is the same as reading records when updating data in a direct access data file. However, when you specify a read-only operation, you cannot write data back to the data file (a CTL DOMAIN ERROR is generated); this prevents data from accidently being written to the data file. You specify the read-only operation as follows:

```
CTL ← 'INR [device/file number] [ID = (file ID)] [MSG = OFF] [TYPE = N]'
```

Read-only data from a direct access data file.

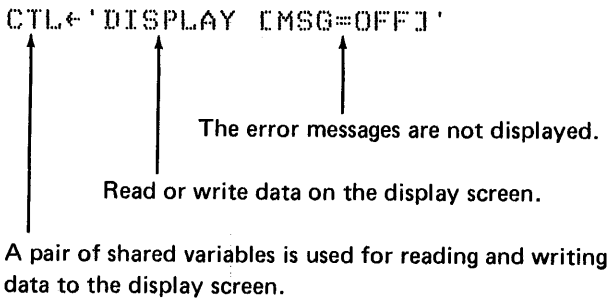
For example:



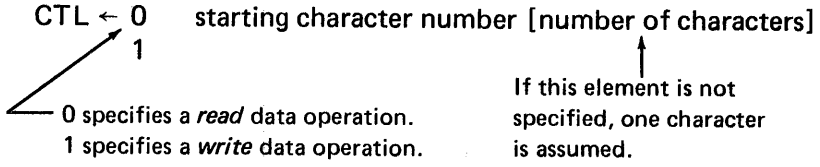
See *To Update a Direct Access Data File* for a description of how records are read from a direct access data file.

To Read Data from and Write Data to the Display Screen

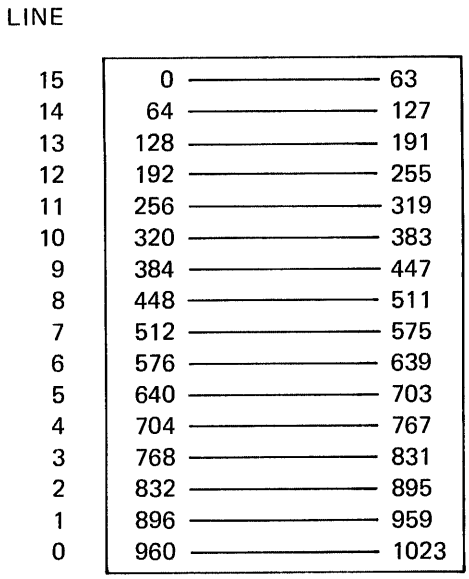
Reading and writing data on the display screen is similar to reading and writing data on a direct access data file. That is, the operation requires a pair of shared variables, with one of the shared variables having the prefix CTL and the other having the prefix DAT. You specify the operation as follows:



Once the operation is specified, the CTL shared variable is used to specify what character positions to read or write. To do this you must assign a two- or three-element vector to the CTL shared variable, as follows:



Note: Each character position on the display screen is considered one record. The character positions are numbered as follows:



For example:

```
CTL←0 20 10
```

Read 10 characters starting with character position 20.

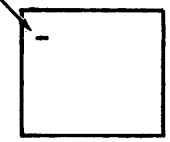
When the CTL shared variable specifies that characters be read, the cursor appears on the display screen at the character position specified by the starting character number. You can then modify the information on the display screen for the specified number of character positions. Also, the insert, delete, and ATTN key perform the same functions within the specified number of character positions as they do during standard APL keyboard input. Then, when the EXECUTE key is pressed, the specified characters are read from the display screen and assigned to the DAT shared variable as a character vector.

When the CTL shared variable specifies that characters be written, the data currently assigned to the DAT shared variable is written to the specified positions on the display screen. DAT must be a character vector with at least as many characters as specified by the CTL shared variable.

In each case, after reading or writing records on the display screen, the I/O processor assigns a return code to the CTL shared variable.

Following is an example of reading and writing data on the display screen:

```
      1 ⎕SVO 2 7⍴'CTLDISPDATDISP'
2 2

      CTLDISP←'DISPLAY'
      CTLDISP
0 0

      CTLDISP←0 64 64
      CTLDISP
0 0
      ⍴DATDISP
64
      DATDISP
DISPLAY I/O


      DATDISP←1024⍴DATDISP


      CTLDISP←1 0 1024
```



The cursor appears on line 14.

Now, enter DISPLAY I/O and press EXECUTE.

After the data is assigned to the DAT shared variable, the DAT shared variable can be used like any other variable.

Write the value currently assigned to DAT on the display screen.

The display screen now looks like this:

```
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
DISPLAY I/O
```

Now, scroll up 2 lines
and enter

```
      CTLDISP←⍳0
      CTLDISP
0 0
```

to terminate the operation.

To Send Data to the Printer

You can control what information is printed by using an APL shared variable. You specify the operation as follows:

```
SV← 'PRT [MSG = OFF]'
      ↑
```

The shared variable is used for printing data.

For example (assume SV is already established as a shared variable):

```
      SV←'PRT MSG=OFF'
      SV
0 0
```

Once the operation is specified, only the information (a character scalar or vector) assigned to the shared variable is printed. The I/O processor assigns a return code to the shared variable after each print operation. For example:

```
      SV←'ONLY THE DATA ASSIGNED'
      SV
0 0
      SV←'TO THE SHARED VARIABLE IS PRINTED'
      SV
0 0
```

When using a shared variable for print operations, you can only assign character scalars or vectors to the shared variable. However, you can also print a matrix as follows:

```
      MATRIX←4 2⍴'12345678'
      MATRIX
12
34
56
78
      CR←⎕AVC[15 7]
      SV←¯1↓,(⍉MATRIX),CR
      SV
0 0
      SV←⍳0
      SV
0 0
```

Carriage return character (assuming ⎕IO←1).

This expression catenates a carriage return character at the end of each row of the matrix and then ravels the matrix into a vector, as follows:

1 2 CR 3 4 CR 5 6 CR 7 8 ~~CR~~ ← The last CR is dropped.

Terminate the operation.

Now, when the vector is printed, a new line is started each time the carriage return is encountered, as follows:

```
1 2
3 4
5 6
7 8
```

TERMINATING THE OPERATION AND RETRACTING THE SHARED VARIABLE OFFER

As discussed previously, you can terminate an input/output operation by assigning an empty vector to the appropriate shared variable. After you terminate the operation, the I/O processor assigns a return code to the shared variable, and then the shared variable can be used to specify another input/output operation.

See the *IBM 5110 APL Reference Manual* for more information on assigning an empty vector to the shared variable.

There are four other ways that you can terminate the operation:

- Use the ⎕SVR system function to retract the shared variable offer.
- Use the ⎕EX system function to expunge the shared variable name.
- Complete execution of a user-defined function in which the shared variable is made local to that function.
- Use the)ERASE system command to erase the shared variable name.

In each case, the operation is terminated and the shared variable offer is retracted. However, a return code is not assigned to the shared variable to indicate whether or not the operation was terminated successfully.

Generally, the ⎕SVR function is used after the operation is terminated and the shared variable is no longer required for any input/output operations. The ⎕SVR system function requires one argument; this argument must be character data that represents the shared variable names being retracted. If more than one name is required, the names may be entered as a character matrix with each row representing an individual name. For example:

```
⎕SVR 'A'
```

The variable name A is no longer a shared variable.

```
⎕SVR 3 1⍴'ABC'
```

The variable names A, B, and C are no longer shared variables.

The ⎕SVR function generates an explicit result of 2 for each shared variable offer with the I/O processor that is successfully retracted. For example:

```
      X←⎕SVR 3 1⍴'ABC'
      X
2 2 2
```

In this case A, B, and C are shared variables.

Once the □SVR function is used to retract a shared variable offer, the shared variable becomes an ordinary variable.

```
      A←3+4
      A
7
```

The following chart summarizes the data file types for input/output operations. Sequentially accessed data files require one shared variable, and directly accessed data files require a pair of shared variables (CTL/DAT). Records in the data files can be blocked and spanned or unblocked and unspanned. Blocked and spanned records can span sector boundaries. Unblocked and unspanned records cannot span sector boundaries; that is, the record must be less than or equal to the sector size and there is only one record per sector.

TYPE= Parameter and Data Format	**File Type Displayed Using the)LIB Command**		**Comments**
	Sequential (OUT)	**Direct (OUTF)**	
A APL internal blocked/spanned	8	10	• The 5110 defaults to TYPE=A if the parameter is not specified. • For file type 10, all the records in the file must be the same size and representation (character, binary, fixed point, or floating point).
I Character data blocked/spanned	2	9	• File types 2 and 9 can be read sequentially using an IN operation. • File type 9 (blocked and spanned) uses less storage than file type 10 for character data. • For file type 9, all the records in the file must be the same size.
U Character data unblocked/ unspanned		9 or B9	• File types 9 and B9 can be read sequentially using an IN operation. • File type B9 is the basis of exchange with other products. • All the records in the file must be the same size.
M APL internal blocked/spanned	8	15	• For file type 15, all the records in the file must use the same or less storage than the first record written to the file, and records must be read or written one at a time.

```
            **  SEQUENTIAL I/O  **

      Sequential files on diskette are essentially the same
      as tape files.  The two types are:
            GENERAL EXCHANGE (File type 2)
            APL INTERNAL (File type 8)

      Sequential files require only one shared variable.

    1 ⎕SVO 'SEQDSK'
2

      This example specifies 'OUT' to indicate output
      and defaults to the APL INTERNAL File Type.

    SEQDSK←'OUT 11001 ID=(TEST)'
      Once the operation is established and checked
      for valid completion with a '0 0' return code,

    SEQDSK
0 0

      records may be written to the diskette file.
      An integer record is written first.

    SEQDSK←⍳10


      Always check the return code on output.
    SEQDSK
0 0

      APL INTERNAL sequential files permit writing of
      any type of data within the same file. For
      example, you can write a floating point record,

    SEQDSK←.5+⍳10
    SEQDSK
0 0

      and a binary record,

    SEQDSK← 15⍴0 1 0 1 0 1 1
    SEQDSK
0 0

      and a character record.

    SEQDSK←'This is the LAST record.'

    SEQDSK
0 0
```

```
      You can now close the file,

      SEQDSK←⍳0

      SEQDSK
0 0

      and read what was written to the diskette.

      NOTE that only the ID (or Device/File Number) is
      needed to specify a data file on the diskette.

      SEQDSK←'IN ID=(TEST)'

      Check the return code.

      SEQDSK
0 0

      and read each record in the order that it
      was written.

      First, the integer record,
      SEQDSK
1 2 3 4 5 6 7 8 9 10

      the floating point record,
      SEQDSK
1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.5

      the binary record,
      SEQDSK
0 1 0 1 0 1 1 0 1 0 1 0 1 1 0

      and, finally, the character record.
      SEQDSK
This is the LAST record.

      There will now be an empty vector returned,
      which indicates the end of data or an I/O error.

      SEQDSK

      Check for an error condition.

      SEQDSK
0 0
      A '0 0 ' return code indicates that there
      is no more data.
```

GENERAL EXCHANGE (File type 2) files are handled the same as APL INTERNAL files except that all data must be in character format. This type of file may be exchanged with the BASIC Language.

Both GENERAL EXCHANGE and APL INTERNAL format files may be written to tape in a similar fashion and both are compatible with the IBM 5100.

In addition, The 5110 will read 5100 file types 1 and 3.

```
      1 ⎕SVO 'CTLDSK'
2

      1 ⎕SVO 'DATDSK'
2
```

Two shared variables are offered which can be used together for record input/output.

```
      CTLDSK←'OUTF 11001 ID=(TEST)'

      CTLDSK
0 0
```

Assignments to CTLDSK write records onto the diskette. The records must all be the same length and internal representation since these are default TYPE=A records.

```
      CTLDSK←100⍴'A'

      CTLDSK←100⍴'B'

      CTLDSK←100⍴'C'

      CTLDSK←100⍴'D'

      CTLDSK←100⍴'E'

      CTLDSK←100⍴'F'


      CTLDSK← 20⍴'G'
IO STATUS: INVALID DATA TYPE

      CTLDSK← 150⍴'H'
IO STATUS: INVALID DATA TYPE

      CTLDSK←100⍴1
IO STATUS: INVALID DATA TYPE
```

NOTE that the assignment must be character and the length must be 100 since that is true with the first record. The error messages were displayed because the MSG=OFF parameter was not specified.

```
      CTLDSK←⍳0

      CTLDSK
0 0
```

The output operation is now terminated.

Normally, you would want to check return codes to assure proper operation. For example, 1⍆RCODE should always be 0 with record input/output

```
      CTLDSK←'IOR 11001 ID=(TEST)'
```

The file is now being specified for input or output operations.

You could have used INR----to READ ONLY or IORH----to READ or WRITE.

IOR and IORH differ in storage requirements and performance. They are the same in function.

```
      CTLDSK
0 6
```

NOTE that there are 6 records available.

All reads and writes must address one of these actual records.

To add records to a record I/O file originally created using OUTF, you must specify an ADD operation.

CTLDSK will now be used to read and write and DATDSK is the variable containing the data.

```
      CTLDSK←A B C
             ↑ ↑ ↑____NUMBER OF RECORDS TO READ OR WRITE
             ↑ ↑_____FIRST RECORD TO READ OR WRITE
             ↑_______READ OR WRITE (READ=0 AND WRITE=1)


      CTLDSK←0 0
```

This says to read the first record (RECORD 0) and read only one record which is the default for the number of records

```
      ⍴DATDSK
100
```

```
      DATDSK[1 2 3 4 5]
AAAAA

      CTLDSK←0 1 3
          You have read 3 records starting with the second one.

      ⍴DATDSK
3 100
      DATDSK[;1 2 3 4]
BBBB
CCCC
DDDD

      DATDSK←100⍴'Z'
      CTLDSK←1 1

          Now you have written a new second record containing
          100 Z'S.  To check you can reread the first three.

      CTLDSK← 0 0 3
      ⍴DATDSK
3 100

      DATDSK[;1 2 3 4]
AAAA
ZZZZ
CCCC

      CTLDSK←⍳0

          This terminates the input/output operations.
      CTLDSK
0 0
```

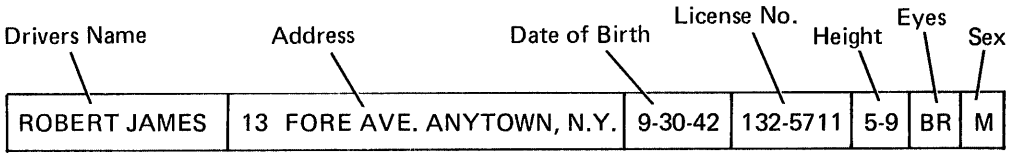
USING THE)RESUME COMMAND

There might be times when you are using a user-defined function to do I/O operations and you want to stop the I/O operation until a later time. If you suspend the user-defined function by pressing ATTN once (a weak interrupt) and then write the contents of the active workspace to the media using the)CONTINUE command, you can use the)RESUME command to load the stored workspace into the active workspace and reestablish the system as it was. That is, the shared variables and suspended function are reestablished in the active workspace, and, if you enter → ⎕LC the user-defined function continues execution from the point where it was suspended.

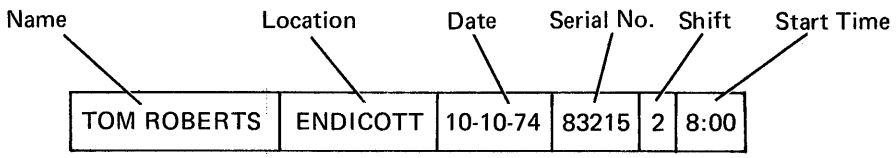
Note: When you use the)RESUME command, you must make sure that the tapes or diskettes are correctly positioned. See the *IBM 5110 APL Reference Manual* for a complete description of the)RESUME command.

MORE ABOUT RECORDS AND FILES

The basic unit of organized data is the *record*. A record is a collection of related data items that are treated as a unit. For example, the driver's license most of us carry is a record. A time card is also a record. Each record contains items related to the purpose of the record. The related items are called *fields*. The following illustration shows a record containing the fields of information that might be found on a driver's license:

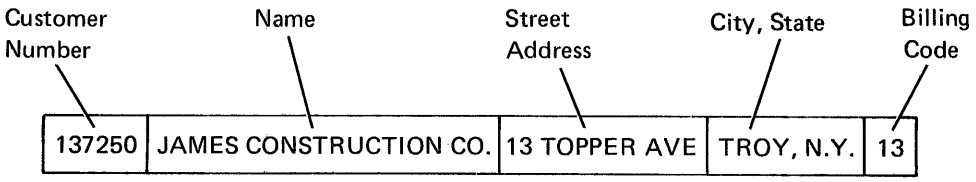


Each field in the previous record contains information relating to a specific driver. The length of the field is the maximum number of characters that is to be placed in the field. The next illustration shows a record containing the fields of information that might be found on a time card:



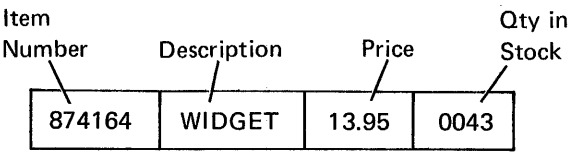
A group of records make up a file. Just as a filing cabinet contains a number of records in some specific sequence, a 5110 data file also contains records in some specific sequence.

The following illustration shows a record containing customer information that would be used in making out an invoice:



The file would contain as many records as there are customer numbers. A file should be given a unique name so that the file can be distinguished from other files. Because the record in the previous illustration contains customer master information, the file could be named CUSTOMER.MASTER. A file containing master information about the products in your inventory could be named ITEM.MASTER.

Different files can contain different record layouts. For example, the following illustration shows a record that has items related to the item file:



Organizing a Diskette File

An important part of any data processing job is *file organization.* File organization is the arrangement of records in a file.

Sequential Access Files

Sequential access files are processed consecutively. For example, an employee master file contains information needed for various reports concerning each employee, such as payroll checks. Because checks are usually processed in order by employee number, records are processed in order. The lowest employee number is processed first and so on until the last record, the highest employee number, is processed.

Sequential processing means records are processed one after another in the order they occur in the file. To process only certain records all records must be processed, or at least read up to the last record to be processed.

Direct Access Files

Since sequential processing can be time consuming, it would be helpful if diskette records were available like books in a library. That is, you could go to an index, find the location where the book is stored, go to the right shelf, and get the book you want. No one would read all the books in the shelves before reading the desired book. Likewise, it would be desirable to skip the records not needed in a job and process only the desired ones. Because direct access files allow specified records to be processed by record number, the limitation of sequential processing can be overcome by an index.

To create and use an index, you could use the following procedure:

1. As you create the direct access, you also create a matrix with each row representing a key field in the record. For example, the first 7 characters in a record are the item number. As you write a record to the file, you also add the item number to a matrix of item numbers.

2. Now, when you want to directly access a record by record number, you can find the record number by using the following user-defined function (assuming ⎕IO←0):

```
      ∇Z←LIST INDEX ITEM
[1]   Z←((LIST[;]=(¯1↑⍴LIST)↑ITEM))/⍳1↑⍴LIST
∇
```

   The explicit result of this user-defined function is the row of the matrix that contains the specified item number. This explicit result also identifies the record number of the specified item number in the data file.

3. Once you find the record number of the specified item number, you can directly access and process that record without regard to its relation to other records.

Ordered and Unordered Records

The records in a direct access file can be *ordered* or *unordered*. An ordered file means that the records are arranged in order according to some major control field or by frequency of use. An unordered file means that the records are not in any particular order.

With a 5110, it takes *less* time to specify and read 100 records at one time than to specify and read individual records 100 times. Therefore, arranging your records in order of frequency of use might save you processing time. For example, a wholesale distributor organizes the file of inventory items by frequency of use. Thus, the most active items are at the beginning of the file. Then, when the file is used to write customer orders, most of the records are located in a small area of the file and can be processed as multiple records. In this example, the total time to process the orders is less than if the records were scattered throughout the entire file.

Maintaining Diskette Files

Once a file is created, *file maintenance* is often necessary. File maintenance means performing those activities that keep a file current for daily processing needs. Some file maintenance activities are: *adding, deleting,* and *updating* records. Adding means putting a record in a file after the file is created. Deleting means identifying a record so it will not be processed with other records. Updating means adding or changing some data in a record.

Adding Records

Records can be added at the end of a file after a file has been created. Thus, the file is extended by the added records.

Sometimes, however, the new records must be merged between the records already in the file. This might be necessary to keep the file in a particular order when the control fields of the new records are not higher in sequence than those already in the file; for example, when you are using the file for search by key operations. To put the new records in the proper sequence, you must use an APL user-defined function to sort the file and create a new file containing the added records or use the Diskette Sort feature if installed. See the *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311, for more information on the Diskette Sort feature.

Tagging Records for Deletion

When a record becomes inactive, you might not want to process it with the other records. A record cannot be physically removed from the file during regular processing; therefore, it is necessary to identify or tag the record so it can be bypassed. One way to tag such a record is to put a code, called a delete code, in a particular location in the record. When the file is processed, your user-defined function can check for the delete code; if the code is present, the record is bypassed.

When several records in a file have been tagged for deletion, you should remove them from the file. This will free diskette space. You can remove the deleted records by using a user-defined function to copy the records to be retained onto another file or write new records over the deleted records.

Updating Records

When you update records in a file, you can add or change some data on the record. For example, in an inventory file you might want to add the quantity of items received to the previous quantity on hand. The record to be updated is read into storage, changed, and written back in its original location.

Designing a Record

The applications that use a certain file determine what data is needed in a record. You should study these applications and then decide the *layout* of the record. Layout means the arrangement of fields in a record. When you design a record, you must consider processing requirements of the record and then determine field length, location, and name.

A name and address file is used to illustrate these design considerations. Each record in the file contains the following data:

Field	*Size (number of characters)*
Customer Number	8
Name	20
Street Address	20
City and State	20
Record Code	2
Delete Code	1
(Other fields)	(total) 47
	Total 116

Determining Field Size

Field size depends on the nature of the data in the field. First, the length of the data may vary. In the example, name is 20 characters. The length of each customer's name varies, but 20 characters should be sufficient for all names. Secondly, all data in a field might be the same length. For example, customer number is eight positions, and all eight positions are used in each record.

There are no firm rules for determining field size. The major problem involves fields with variable length data. For example, if a name is planned as 15 characters, and a new customer has 19 characters in his name, a problem arises when you add his record to the file. To avoid this problem, try to estimate the largest length of data that will be contained in a field. Use this length to determine the field size.

Providing for a Delete Code

Remember that records are not automatically deleted. You might want to place a delete code in a record, and then when the file is processed you must check for the delete code. In the example, if a customer becomes inactive, we do not want to process his record. Thus, a one-position field is included to provide for a delete code.

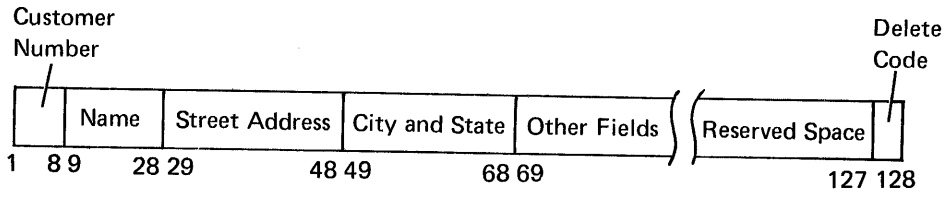
Of course, if you are using an index to find the records in a direct access file, you do not need a delete code. Instead, you can remove the record key from the index. See *Organizing a Diskette File* in this chapter for information on creating an index.

Providing Extra Space

At this stage in planning, it is often wise to allow for data to be added to a record. For example, suppose the name and address file was created with the fields described, and at a later time each customer's zip code is needed. If all positions in the record are used, there is no place to add the zip code. Because the record length is not yet established, we can allow for such additions to this record. Although it is often difficult at the planning stage to imagine what data might be added, it is wise to reserve extra space. A minimum of 10% extra space is suggested.

Documenting Record Layout

When record layouts are documented, your APL user-defined function might be easier to write. A record layout should include the order of the fields in the record, the length of each field, and the name of each field. The following illustration shows the layout of a customer master record:



In the previous example, the sum of the fields is 116 positions. However, the record size is 128 positions, thus reserving 12 positions for data that might be needed at a later time.

Determining the Number of Records in a File

When determining the number of records in a file, you should consider expansion for a reasonable time into the future (at least six months). Then when you create the file, if you place dummy records in the file, these dummy records can then be replaced with valid records at a later time. Of course, you can also add records to the file using an ADD operation.

SUSPENDED FUNCTION EXECUTION

The execution of a user-defined function can be interrupted (suspended) in a variety of ways: by an error message, by pressing the ATTN key, or by using the stop control vector (S∆). In any case, the suspended function is still considered active, since its execution can be resumed. Whatever the reason for the suspension, when it occurs, the statement number of the next statement to be executed is displayed. A branch to the statement number that was displayed or a branch to □LC(→□LC) causes normal continuation of the function, and a branch out (→0) removes the function.

When a function is suspended, the 5110 will:

- Continue to execute system commands except)SAVE,)COPY, and)PCOPY.
- Resume execution of the function at statement n when →n is entered.
- Reopen the definition of any function that is not pendent. A pendent function is a function that called the suspended function. If a function called a function that called a suspended function, it is also pendent (see *State Indicator*).
- Execute other functions or expressions.
- Execute the suspended function again.

Note: The display of output generated by a previous statement might have been interrupted when the suspension occurred. This would be caused by the delay between execution of the statement and display of the output.

STATE INDICATOR

The state indicator identifies which functions are suspended (*) and at what point normal execution can be resumed. Entering)SI causes a display of the state indicator. Such a display might have the following form:

```
      )SI
H[7]  *
G[2]
F[3]
```

This display indicates that execution was halted just before statement 7 of function H, that the current use of function H was invoked in statement 2 of function G, and that the use of function G was invoked in statement 3 of F. The * appearing to the right of H[7] indicates that function H is suspended; the functions G and F are said to be pendent.

During the suspension of one function, another function can be executed. Thus, if a further suspension occurred in statement 5 of function Q, which was invoked in statement 8 of G, a display of the state indicator would be as follows:

```
      )SI
Q[5]  *
G[8]
H[7]  *
G[2]
F[3]
```

An SI DAMAGE error indicates that a suspended function has been edited or a pendent function has been erased and the normal execution of the suspended function can no longer be resumed. When an SI DAMAGE error occurs, the state indicator display will include the damaged function name and the statement number ¯1. For example, if function Q is edited and the modification causes an SI DAMAGE error, the display of the state indicator would be as follows:

```
      )SI
Q[¯1]  *
G[8]
H[7]  *
G[2]
F[3]
```

You can clear a suspension by entering a branch with no argument (that is, →). One suspended function is cleared at a time, along with any pendent functions for that suspended function. The first branch clears the most recently suspended function, as in the following example:

```
      →
      )SI
H[7]  *
G[2]
F[3]
```

It is a good practice to clear suspended functions, because suspended functions use available storage in the active workspace. Repeated use of → clears all the suspended functions; as the functions are cleared, they are removed (cleared) from the state indicator. When the state indicator is completely cleared, the state indicator display is a blank line.

Note: To display the state indicator with local names, enter the)SINL command.

Appendix A. 5110 Compatibility with Other APL Systems

The follwing user-defined functions are example functions that can be used to check 5110 user-defined functions for compatibility with other APL systems:

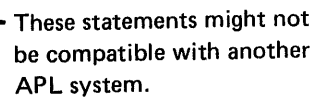
```
      ∇CHECK[⎕]∇
    ∇ CHECK F;EA;RL;CF;FL;ED;⎕IO;⎕PW
[1]   ⍝CHECK FOR POSSIBLE INCOMPATIBILITES IN FUNCTION F
[2]   ⍝EA=ERROR ARRAY          CL=COLUMN LEDGEND
[3]   ⍝RF=ROW FLAGS (ERRORS)        RL=ROW LEDGEND
[4]    ⎕IO←1
[5]    ⎕PW←132
[6]    EA←PORTABLE CF←⎕CR F
[7]    RL←'[', 0 1 ↓(⍕((1↑⍴EA),1)⍴¯1+⍳1↑⍴EA),']'
[8]    RF←(∨/EA)\'*'
[9]    CF←RF,' ',RL,' ',CF
[10]   FL←' ↑'[1+EA]
[11]   ED←(2 1 ×⍴CF)⍴ 2 1 3 ⍉(CF),[0.5](-⍴CF)↑FL
[12]   (~∧/ED=' ')/[1] ED
    ∇
```

```
      ∇PORTABLE[⎕]∇
    ∇ Z←PORTABLE CF;CS;⎕IO
[1]   ⍝RETURN A LOGICAL MATRIX (ONES AND ZEROS) THE SAME SHAPE AS
[2]   ⍝''⎕CR F''--A 1 INDICATES CHANGES REQUIRED FOR PORTABILITY
[3]   ⍝CS=LEGAL CHARACTER SET          CF=CHAR MATRIX OF F
[4]    ⎕IO←1
[5]    CS←⎕AV[(14+⍳9),(25+⍳54),(86+⍳71),159,160,161]
[6]    Z←~CF∈CS
[7]    Z←Z∨(CF='⎕')
[8]    Z←Z∨('1 ⎕SVO' WHEREIN CF)
[9]    Z←Z∨('⎕CC' WHEREIN CF)
[10]   Z←Z∨('⎕AI' WHEREIN CF)
[11]   Z←Z∨('⎕TS' WHEREIN CF)
[12]   Z←Z∨('⎕TT' WHEREIN CF)
[13]   Z←Z∨('⎕UL' WHEREIN CF)
[14]   Z←Z∨('⎕DL' WHEREIN CF)
    ∇
```

```
      ∇WHEREIN[⎕]∇
    ∇ Z←A WHEREIN B;⎕IO
[1]   ⍝LOCATION OF VECTOR A IN ARRAY B
[2]    ⎕IO←0
[3]    Z←((-⍴⍴B)↑1)↓∧/[0](⍉((¯1↓⍴B),⍴A)⍴⍳⍴A)≡0,(A←,A)∘.=B
    ∇
```

```
      ∇TESTFUNCTION[⎕]∇
    ∇ TESTFUNCTION
[1]     5 ⎕CC 20
[2]     1 ⎕SVO 'A'
    ∇
```

```
      CHECK 'TESTFUNCTION'
  [0] TESTFUNCTION
* [1] 5 ⎕CC 20
       ↑
* [2] 1 ⎕SVO 'A'
       ↑
```



These statements might not be compatible with another APL system.

READER'S COMMENT FORM

Please use this form only to identify publication errors or request changes to publications. Technical questions about IBM systems, changes in IBM programming support, requests for additional publications, etc, should be directed to your IBM representative or to the IBM branch office nearest your location.

Error in publication (typographical, illustration, and so on). **No reply.**

Page Number *Error*

Inaccurate or misleading information in this publication. Please tell us about it by using this postage-paid form. We will correct or clarify the publication, or tell you why a change is not being made, provided you include your name and address.

Page Number *Comment*

Note: All comments and suggestions become the property of IBM.

Name _____

Address _____

_____

● No postage necessary if mailed in the U.S.A.

Cut Along Line

Fold

Fold

FIRST CLASS
PERMIT NO. 40
ARMONK, N. Y.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Development Laboratory
Publications, Dept. 245
Rochester, Minnesota 55901

Fold

Fold

IBM®

International Business Machines Corporation

General Systems Division
5775D Glenridge Drive N. E.
P.O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
U.S.A.
(International)

IBM 5110 APL User's Guide Printed in USA SA21-9302-1

IBM

International Business Machines Corporation

General Systems Division
4111 Northside Parkway N.W.
P.O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
U.S.A.
(International)

IBM 5110 APL User's Guide Printed in U.S.A. SA21-9302-1