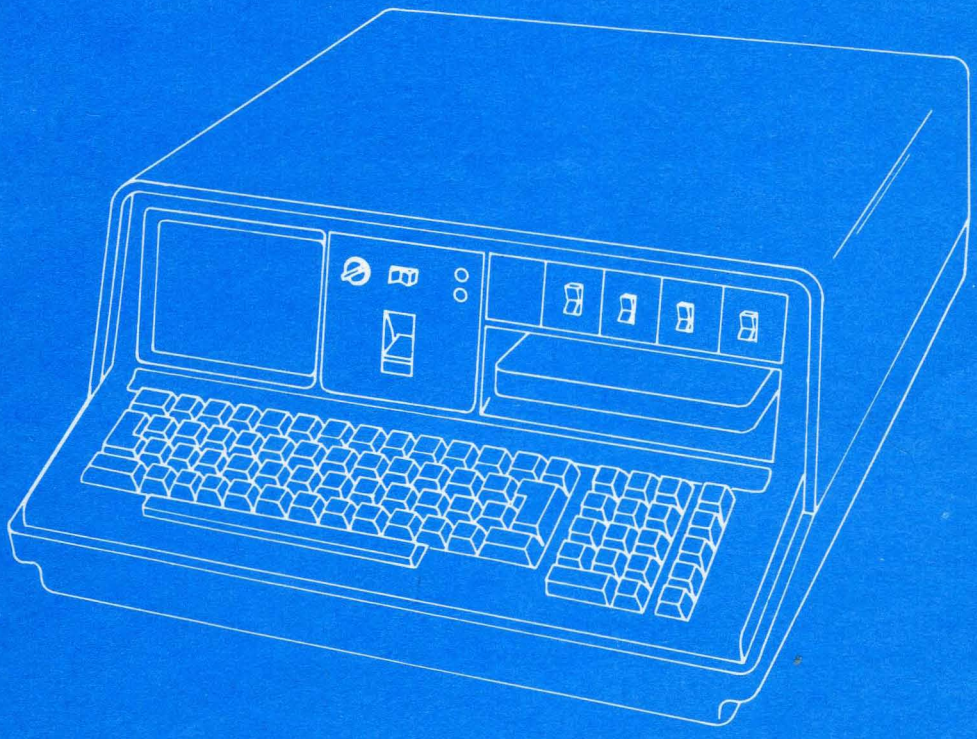
IBM

IBM 5110
BASIC User's Guide

5110



SA21-9307-0

IBM 5110 BASIC User's Guide

First Edition (December 1977)

Changes are continually made to the specifications herein; any such changes will be reported in subsequent revisions or technical newsletters.

Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A Reader's Comment Form is at the back of 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

Before using this manual, you should be familiar with the information in the *IBM 5110 BASIC Introduction*, SA21-9306, such as:

- Entering data from the keyboard
- The arithmetic operators
- How to enter a program
- Using data files
- Using arrays

This manual gives you conceptual information about using the 5110 with the BASIC language and is intended to be used with the *IBM 5110 BASIC Reference Manual*, SA21-9308. The topics covered in this manual include:

- Computer concepts for data processing
- An approach to breaking your application into small parts to make programming easier
- Changing the sequence of execution within your BASIC program
- Formatting the data on the display screen
- Entering uppercase and lowercase characters
- Using a procedure file to replace keyboard input
- Sounding the audible alarm
- Using tape and diskette storage
- Formatting printed reports
- Additional tips and techniques

Since this manual is not intended to give you a complete description of the syntax and rules required for each BASIC command and statement, you must use the *IBM 5110 BASIC Reference Manual* for this description.

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 3, *Changing the Sequence of Execution In Your BASIC Program*, when you need information on program loops.

Prerequisite Publication

IBM 5110 BASIC Introduction, SA21-9306

Related Publications

IBM 5110 BASIC Reference Manual, SA21-9308

IBM 5110 BASIC Reference Handbook, GX21-9309

IBM 5110 Customer Support Functions Reference Manual, SA21-9311

Chapter 1. 5110 Data Processing Concepts

INTRODUCTION

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?

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 a wide variety of operations. 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 work 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 an employee might process information for the job of billing. Assume for this job that the employee works with the following data:

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

The employee receives a copy of the customer order after the order is shipped. He uses this document to prepare the invoice that he sends to the customer. To prepare the invoice, the employee:

1. Looks up, in a price catalog, the price of each item in the order
2. Multiplies the price of each item by the quantity shipped
3. Adds the total price of each item to get the total amount of the invoice
4. Checks the customer records to see if any special discounts apply, and adjusts the invoice accordingly
5. Types the invoice
6. Adjusts the accounts receivable records to show what the customer owes
7. Updates the inventory files to show the reduced stock

For each invoice he prepares, the employee follows the same procedure. In computer terms, the 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 processing–the invoice and the updated records–are his *output*.

As shown in Figure 1, computer data processing can speed up a billing operation and reduce costly errors. Data (customer order information) can be entered at high speed via the keyboard; many records can be quickly referenced and updated in a magnetic storage medium (tape or diskette); the processing unit can store and carry out instructions (a program) and perform needed calculations; and a printer can print the invoice.

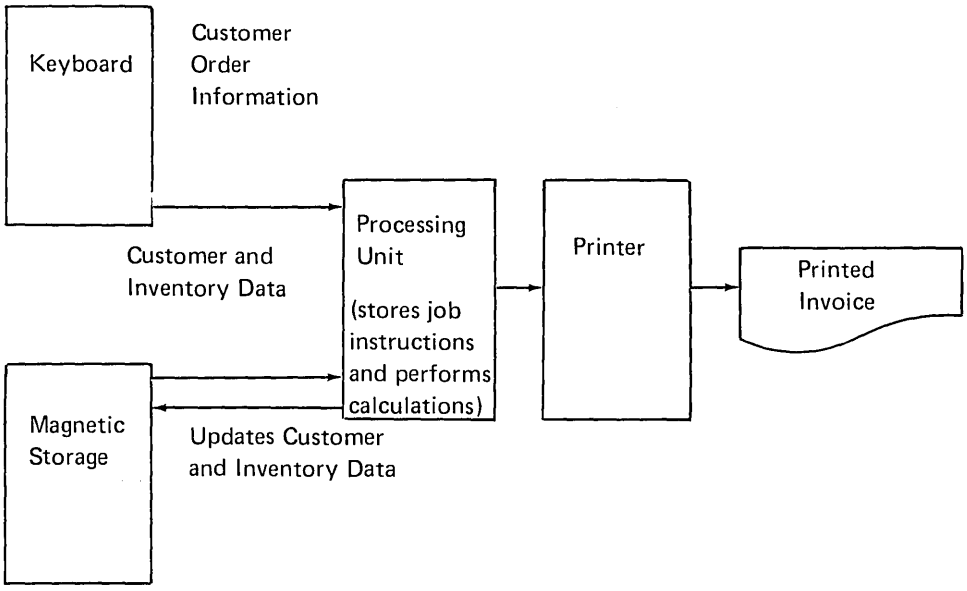


Figure 1. Computer Data Processing

The 5110 Model 1 Computing System (Figure 2) contains the following elements, which are components of the data processing system:

- Input Elements–keyboard, tape, diskette
- Output Elements–tape, diskette, printer, display screen
- Processing Elements–main storage, tape, diskette, programs

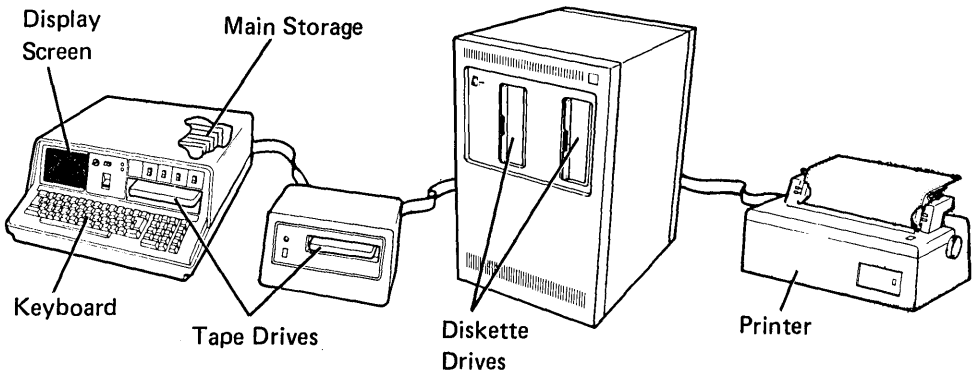


Figure 2. 5110 Computing System Data Processing Elements

The keyboard is the device the operator uses to key (enter) data into the processing unit.

The tape and diskette are used either as input or output devices. Input data or programs can be entered into the system using the tape or diskette. Output data can be stored on the tape or diskette for use in other programs.

The printer records on paper (prints) the data sent to it by the processing unit. This printed material is sometimes referred to as the hard copy output.

The display screen displays output data sent to it by the processing unit. The system uses the display screen to communicate with the operator by displaying information keyed on the keyboard so that the operator can verify the information before it enters the system. The system also displays messages that identify keying errors and provide operator guidance or specific processing information as required.

The processing unit is made up of the main storage, tape or diskette, and programs. The tape or diskette is used to store files of information and programs that are used by the system. Main storage is the part of the processing unit that holds a program so that the system can execute the steps in the program. Data is moved from tape or diskette into main storage for use by the program being executed.

ADVANTAGES OF COMPUTER DATA PROCESSING

If data processing is always a matter of input, processing, and output, how is computer data processing different from manual or mechanical data processing? Computers process large volumes of data faster, more accurately, in less space, and with greater versatility.

- Speed. Because computers process data electronically, they operate at remarkable speeds that save a tremendous amount of time.

- Accuracy. A computer does exactly what it is told to do and only what it is told to do. Because of this constant dependence on instructions, a computer follows program after program, second after second and hour after hour, with unvarying accuracy.

  Computers also reduce transcription errors by dramatically reducing the need for manual transcription. Once you record data on a tape or diskette that data may never have to be written by hand again–you can update as many different customer records, complete as many different kinds of forms, and create as many different reports from that data as you have application programs that use the data. By referring to the display screen while first recording the data, you can insure that the data is keyed correctly. Programs that use the data can perform control total checks and balances to continually validate the accuracy of the data.

- Data Compression. Computers miniaturize data. Suppose a business enters its accounts receivable transactions in a machine-posted register like the one shown below.

ACCOUNTS RECEIVABLE TRANSACTION REGISTER

07/11/-- PAGE 001

DATE	CUST NO	CUSTOMER NAME	JOURNAL NO	INVOICE NO	CASH AMOUNT	INVOICE AMOUNT	JOURNAL AMOUNT
07/11/--	759820	SOUND OF THE SEVENTIE		063420		$ 46.23	
07/11/--	633870	OLDE VILLAGE SHOPPE		063421		89.70	
07/11/--	642990	PARAGON TV SALES		063422		20.30	
07/11/--	122620	CANNIZONI STUDIOS		063423		129.76	
07/11/--	682030	RAYMONDS RAPID REPAIR			$ 63.80		
07/11/--	742950	SARATOGA VARIETY			29.72		
07/11/--	014280	BAKER BRADLEY & CO.			43.50		
07/11/--	872060	UNIVERSITY ELECTRIC			97.75		
07/11/--	883290	VILLAGE MUSIC & TV	07-036				$18.23CR
07/11/--	006280	ALLSTONS	07-037				10.70CR
		TOTALS			$234.77*	$285.99*	$28.93CR*

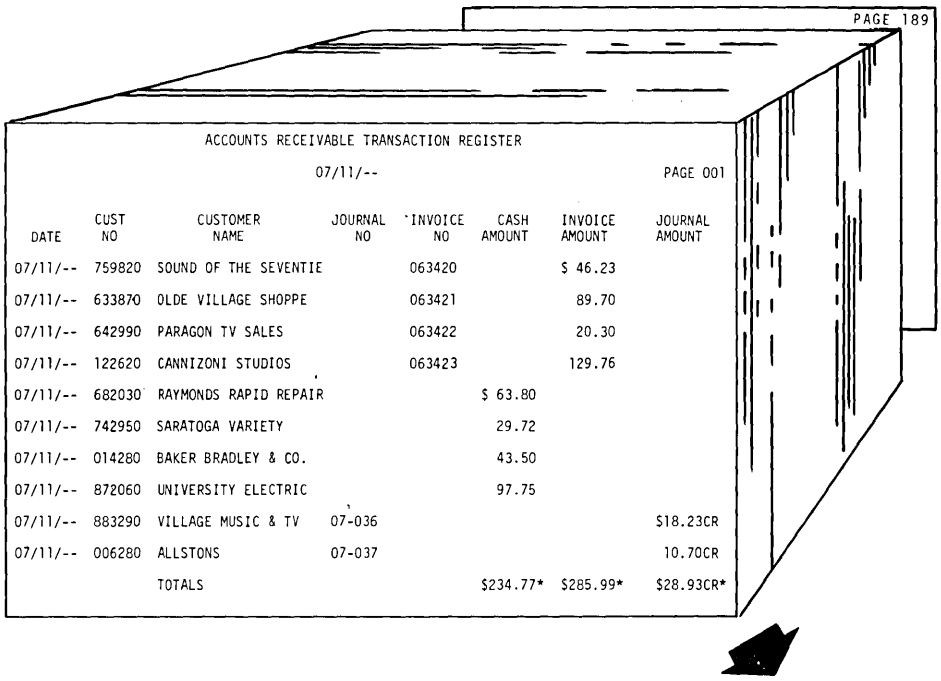
The preceding example shows 10 sample entries, or records. Several thousand such transaction records can be stored by the system on one diskette. That is, the system enables you to store large volumes of business information in an economical and manageable form that can be processed by a machine.

- Versatility. The number of different tasks a computer can do is limited only by the number of different programs run on it. The computer can do much more than just add, subtract, multiply, and divide. The 5110 can, for example, prepare invoices, keep accounts receivable up to date, print weekly paychecks, and analyze data for thorough cost and sales analysis.

Speed, accuracy, data compression, and versatility combine to reduce data processing errors and increase productivity. But a less obvious advantage of computers has a more fundamental effect.

Computers impose discipline. As explained, a computer is helpless without programs–it cannot think for itself. Neither can a computer guess whether its programs really reflect the problems at hand–you must see that they do. In other words, you must carefully analyze the data processing requirements of your organization in order to take full advantage of a computer. For instance, with the data processing capabilities provided by a 5110, what additional cost analysis, inventory control, and auditing procedures would you like to implement in your organization?

The responsibility for analyzing an organization's data processing requirements falls, of course, to management. But the discipline imposed by a computer extends throughout the data processing activities of the organization. Once you've designed or selected computer programs that reflect management directives, you've established management control that is automatically practiced each time those programs are used.

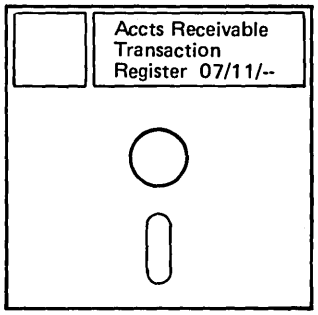


PAGE 189

ACCOUNTS RECEIVABLE TRANSACTION REGISTER

07/11/-- PAGE 001

DATE	CUST NO	CUSTOMER NAME	JOURNAL NO	INVOICE NO	CASH AMOUNT	INVOICE AMOUNT	JOURNAL AMOUNT
07/11/--	759820	SOUND OF THE SEVENTIE		063420		$ 46.23	
07/11/--	633870	OLDE VILLAGE SHOPPE		063421		89.70	
07/11/--	642990	PARAGON TV SALES		063422		20.30	
07/11/--	122620	CANNIZONI STUDIOS		063423		129.76	
07/11/--	682030	RAYMONDS RAPID REPAIR			$ 63.80		
07/11/--	742950	SARATOGA VARIETY			29.72		
07/11/--	014280	BAKER BRADLEY & CO.			43.50		
07/11/--	872060	UNIVERSITY ELECTRIC			97.75		
07/11/--	883290	VILLAGE MUSIC & TV	07-036				$18.23CR
07/11/--	006280	ALLSTONS	07-037				10.70CR
		TOTALS			$234.77*	$285.99*	$28.93CR*



Chapter 2. Elements of a BASIC Program

Before reading this chapter, you should be familiar with the information in the *5110 BASIC Introduction*, such as:

- Entering data from the keyboard
- The arithmetic operators
- Numeric and character variables
- The arithmetic operator hierarchy
- Entering a BASIC program
- The BASIC language statements REM, INPUT, GOTO, STOP, LET, IF, FOR, NEXT, READ, DATA, RESTORE, PRINT, OPEN, CLOSE, RESET, PUT, GET, and DIM
- The system command used to store and retrieve programs on a tape or diskette

In this chapter, the following topics are discussed:

- Defining a program
- Analyzing an application (problem) so that BASIC programs can be used to process information
- The most commonly used BASIC statements

DEFINING A PROGRAM

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 BASIC. BASIC is a simple-to-use programming language with which you describe how you want the 5110 to do the job.

The next section presents an approach to analyzing an application so that a BASIC program can be used to help process information. This approach helps you break down an application into manageable parts so that you can apply BASIC statements you know can process the information. Breaking down an application into manageable parts promotes thoroughness and allows the application to be solved (programmed) faster.

PROCESSING INFORMATION

Every problem consists of three parts:

- The *input* data required to generate the final result
- The *process* (BASIC statements) required to generate the final results
- The *output*, which is the final result

Each part might consist of one statement or several statements. 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 is the primary reason for a program 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 main storage, 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 need the displayed results to evaluate different plans

Once you have answered these questions, you know the purpose of a program.

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?

For 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 program (for example, headings and labels). There might also be data that is usually constant but, for certain applications, must be changed. This data might be coded in the program 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:

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

Process

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:

$$\text{COMPOUND INTEREST} = \text{PRINCIPAL} \left(1 + \frac{\text{Rate}}{100}\right)^{\text{Years}}$$

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

```
A = 1+R/100
B = A↑Y
C = P*B
```

R is the interest rate
Y is the time in years
P is the principal
C is the compound interest

PUTTING IT ALL TOGETHER

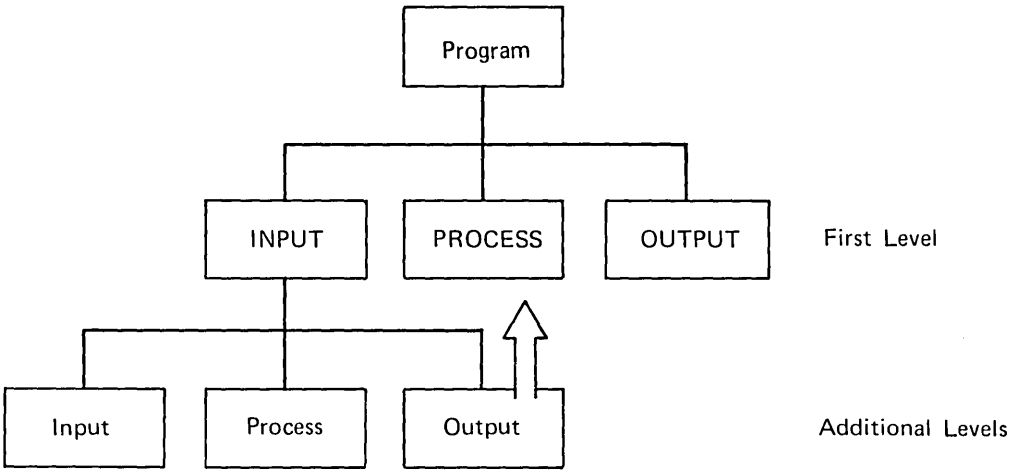
Now that you have considered the three parts of information processing, it is time to write your program. For the compound interest example, your program might look like this:

```
0010 PRINT 'ENTER THE INTEREST, PRINCIPAL, AND YEARS'
0020 INPUT R,P,Y
0030 A=1+R/100
0040 B=A↑Y
0050 C=P*B
0060 PRINT 'THE INTEREST EARNED IS:'
0070 PRINT C-P
```

So far, you have taken a simple application and designed a program 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

For complex or large applications, you might want to break the INPUT, PROCESS, and OUTPUT down into additional levels of input, process, and output.



This allows you to break each first-level INPUT, PROCESS, and OUTPUT part into manageable parts. Let's continue with the compound interest problem and expand the first-level INPUT portion into additional levels of input, process, and output. That is, the INPUT portion is going to be treated 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 500,000.00.

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 program; 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 500,000.00.

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

```
0010 PRINT 'ENTER THE INTEREST RATE, YEARS AND PRINCIPAL'
0020 INPUT I,Y,P
0030 IF I≤18 GOTO 60
0040 PRINT 'THE INTEREST RATE IS GREATER THAN 18 PERCENT'
0050 GOTO 10
0060 IF Y≤40 GOTO 90
0070 PRINT 'THE NUMBER OF YEARS IS GREATER THAN 40'
0080 GOTO 10
0090 IF P≤500000 GOTO 120
0100 PRINT 'THE PRINCIPAL IS GREATER THAN 500,000.00'
0110 GOTO 10
0120 .
     .
     .
```

For complex or large applications, you could also break down the first-level PROCESS and OUTPUT portions; however we are not going to do that for this example.

As you break an application down into manageable parts, you might want to have a separate program for each part. For example:

```
0005 USE I,Y,P
0010 PRINT 'ENTER THE INTEREST RATE, YEARS, AND PRINCIPAL'
0020 INPUT I,Y,P
0030 IF I≤18 GOTO 60
0040 PRINT 'THE INTEREST RATE IS GREATER THAN 18 PERCENT'
0050 GOTO 10
0060 IF Y≤40 GOTO 90
0070 PRINT 'THE NUMBER OF YEARS IS GREATER THAN 40'
0080 GOTO 10
0090 IF P≤500000 GOTO 120
0100 PRINT 'THE PRINCIPAL IS GREATER THAN 500000.00'
0110 GOTO 10
0120 CHAIN 'E80',2
```

The input program is loaded from file 1 and executed.

The CHAIN statement automatically loads the program from file 2 on device E80.

```
0010 USE I,Y,P,C
0020 A=1+I/100
0030 B=A↑Y
0040 C=P*B
0050 CHAIN 'E80',3
```

The process program is loaded from file 2 and executed.

```
0005 USE I,Y,P,C
0010 PRINT 'THE INTEREST EARNED IS:'
0020 PRINT C-P
0030 STOP
```

The output program is loaded from file 3 and executed.

CONCLUSION

When solving a problem using the 5110, break the problem down into manageable parts. To do this, first focus on the program output; this is the primary interface to the user. The output 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 BASIC language to being able to use the BASIC language and then to generating programs that solve real problems.

BASIC STATEMENTS MOST COMMONLY USED FOR INFORMATION PROCESSING

Following is a description of some BASIC statements that you will use for the input, process, and output portions of a BASIC program.

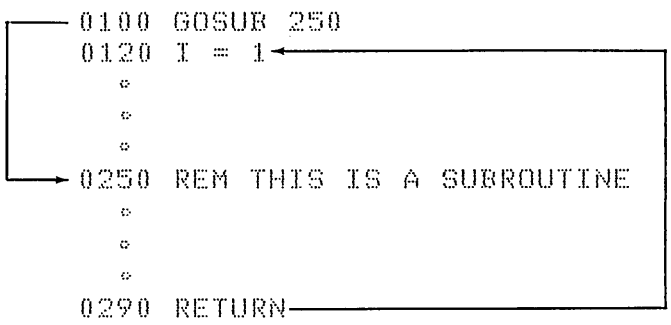
Input Statements

Statement	Description
INPUT	Requests that you enter data from the keyboard during the execution of the program. The data entered from the keyboard is assigned to a corresponding name (variable) specified by the statement.
DATA	Creates a string of numeric and/or character values that can be used by the program. The values are assigned to a corresponding name specified by a READ statement.
READ	Assigns values to variables and arrays from the values created by a DATA statement.
OPEN	Causes specified input and/or output files to be activated. The files can then be used for the input and/or output operations.
CLOSE	Causes the specific input and/or output file to be deactivated. The files cannot then be used for input and/or output operations until they are opened again.
READ FILE	Reads records from a specified record I/O data file (see Chapters 7 through 10) and assigns the data to specified variables.
DIM	Specifies the size of arrays and character variables used by the program.
GET	Reads data from a specified stream I/O data file and assigns the data to specified variables.

Process Statements

Statement	Description
LET	Assigns the value of an expression to a variable.

FOR	Marks the beginning of a loop and specifies the condition of its execution and termination. The NEXT statement is used to mark the end of the loop:
	`0100 FOR K = 1 TO 10`
	`  .`
	`  .`
	`  .`
	`0130 NEXT K`

NEXT	See the FOR statement.
GOSUB	Transfers control to the first statement of a subroutine. Then when a RETURN statement is executed, control returns to the next statement following the GOSUB statement:  A subroutine is useful when the same statements must be executed several times in the program.
RETURN	See the GOSUB statement.
GOTO	Transfers control to a specified statement.
IF	Causes the program action to be determined as the result of the evaluation of a condition.

Output Statements

Statement	Description
PRINT	Causes data to be displayed on the display screen.
PRINT FLP	Causes data to be printed on the 5103 Printer. Data can be formatted as it is being printed if the PRINT FLP statement is used in conjunction with the IMAGE and FORM statements.
WRITE FILE	Adds a new record at the end of a record I/O file.
REWRITE FILE	Updates (rewrites) a record that already exists in a record I/O file.
PUT	Writes data from specified variables to a stream I/O file.

Chapter 3. Changing the Sequence of Execution in Your BASIC Program

In this chapter, the following topics are discussed:

- Using loops to do the same calculations repeatedly
- Using functions or subroutines to do the same operation several times
- Changing to another BASIC program from a program currently being executed

LOOPS

Suppose you want to display credit amounts of $100 to $5000 in increments of $100, along with the monthly finance charge of 18% per year (.015 per month). You could do it simply enough by writing the following statements:

```
0010 PRINT 100, 100*.015
0020 PRINT 200, 200*.015
0030 PRINT 300, 300*.015
0040 PRINT 400, 400*.015
   .
   .
   .

0490 PRINT 4900, 4900*.015
0500 PRINT 5000, 5000*.015
```

Although this technique works correctly, it is time consuming and tedious. In displaying an amount and its finance charge for all amounts from 100 to 5000, what we are really doing is performing the same operation repeatedly, but using different numbers each time. Calculations that are to be repeated can generally be done efficiently by a simple programming device known as a *loop*.

Here's a concise method of performing the same operations shown previously:

```
0010 P=100
0020 PRINT P,P*.015
0030 P=P+100
0040 GOTO 20
```

Here, we have created a loop in statements 20 through 40. When the program is run, the PRINT statement will be executed once each time the value of P increases by 100. The statement that makes the loop possible is the GOTO statement. It alters the normal sequence of execution by directing the computer to execute a different statement. It does this by referring to the number of that statement. The statement GOTO 20 directs the computer back to statement 20, which displays the value of P and its finance charge. Statement 30 then increases the value of P by 100, and statement 40 is executed again, *branching* the program back to statement 20.

There is one problem with the loop we have shown here: there is no provision for ending the loop. Consequently, not only will we get results for values from 100 to 5000, but also for 5100, 5200, and so on, unless we take some action to stop execution. In this program, we want the loop to end after we reach the value 5000, or, put another way, we want the loop to continue as long as P is less than or equal to 5000. To provide this action, we should build into the loop a test from some condition, so that when the condition is met, the loop will end automatically.

Using the IF Statement

An IF statement says it quite concisely:

```
0010 IF P≤5000 GOTO 20
```

This IF statement says that if P is less than (<) or equal to (=) the value 5000, the program is to branch to statement 20. Here we have incorporated the GOTO statements into the IF statement. Let's put this new statement into the program and see what happens:

```
0010 P=100
0020 PRINT P,P*.015
0030 P=P+100
0040 IF P≤5000 GOTO 20
```

As long as P satisfies the condition, P less than or equal to 5000, execution will loop back to the PRINT statement. However, when P no longer satisfies the condition–when P is greater than 5000–the loop will end automatically and the execution will *fall through* the IF statement to the next statement, which in this case is an END statement signifying the end of the program.

The IF statement has many applications, some of which can be quite sophisticated, depending on the condition tested in the statement. For example, conditions such as the following can be tested:

```
0160 IF A=0 GOTO 60
0170 IF A=0 GOTO 60
0180 IF B-X/Y<Z↑2 GOTO 80
```

The first example is quite simple: if the value of the variable A is equal to 0, branch to statement number 60. The second statement tests the same condition as the first statement, but substitutes the word THEN for GOTO. In the IF statement, THEN and GOTO have exactly the same meaning. The third statement makes a test between two sets of expressions. The first expression evaluates B-X/Y. The second expression evaluates Z↑2. If the value of the first expression is less than (<) the value of the second expression, the program is to branch to statement 80.

Relational Operators

The IF statements illustrated in these examples used the symbols <, >, and =. These symbols are part of a set of operators called relational operators. Relational operators are used only in IF statements; they test the relationship between two expressions. It is important to note that relational operators do not perform any arithmetic operations. They simply test whether or not a condition is satisfied. For example, in statement 40, the equal sign does not mean that P is to be given the value 5000; it tests whether the value already assigned to P equals 5000. If a condition is satisfied (if P does equal 5000 in this example), then the condition is considered true. If a condition is not satisfied (if P does not equal 5000), the condition is considered false. Thus, a relational operator says that if the condition being tested is true, the action specified is taken; otherwise, the action is not taken. Reviewing this concept using the example IF A = 0 GOTO 60, if the condition is true (A does equal 0), then the branch to statement number 60 is made; otherwise the branch is not made. Instead, the program continues with the next statement in sequence.

The relational operators and their definitions are:

Operator	Meaning
=	Equal to
< > or ≠	Not equal to
>	Greater than
> = or ≥	Greater than or equal to
< = or ≤	Less than or equal to

Here are some examples:

```
0030 IF A=B GOTO 500
0190 IF 'PRINT'<'PRIZE' GOTO 300
0800 IF A$≠D$ GOTO 5190
```

In the first example, a test is made between the values contained in the arithmetic variables A and B. The second example illustrates comparison of character data. For character data, a comparison is made according to the EBCDIC collating sequence of each character in corresponding positions in the constant. In other words, the first character of one constant is compared to the first character of the other constant, the second compared to the second of the other, and so on. In this example, the first three letters of the constants compare equal, but when the letter N is compared to Z, they compare unequal. The letter N, occurring before the letter Z in the alphabet, registers less than in the collating sequence. At this point, the condition tested would be met; that is, the character string PRINT is indeed less than PRIZE.

In the third example, character variables are compared. Let's assume that the variable A$ contains the value ON and the variable D$ contains ONLY. The first 2 characters match, but when the letter L is compared to a blank, which is assumed for comparison purposes, they do not match. Thus, the result in this case would also be true, because the value of A$ is not equal to the value of D$. If, however, A$ and D$ do contain matching strings, say both contain the characters ONLY, then the test results would be false–A$ and D$ would be equal, thereby not satisfying the condition of the test.

Logical Operators

The example IF A = B tests the relationship between two expressions. Suppose, however, that you wish to take action if more than one relationship is true. For example, suppose that not only must A equal B but also X must equal Y. You could make these comparisons by using the logical AND operator, written as &:

```
0040 IF A=B&X=Y GOTO 100
```

Statement 40 says that if A equals B and X equals Y, then statement 100 is executed. If only one comparison, or neither comparison, is true, program execution continues with statement 60.

The IF statement can specify two logical operators:

Operator	Meaning
&	AND
!	OR

The AND operator states that both conditions of a test must be true for the entire expression to be true; the OR operator states that either condition (or both) must be true for the expression to be true.

If you want to branch to statement 100 if either A equals B or X equals Y, you could write this statement:

```
0050 IF A=B!X=Y GOTO 100
```

Here are other examples of the AND and OR operators:

```
0070 IF C$>D$!J$=K$ GOTO 50
0080 IF A1≠A2&J$>'CAT' GOTO 300
```

The first example tests an OR condition using character variables. It says that if the value in the variable C$ is greater than the value in D$ or if the values in J$ and K$ are equal, then a branch is made to statement 50.

The second example tests an AND condition using mixed variables. It says that if the value in the arithmetic variable A1 is not equal to the value in A2 *and* the value in the character variable J$ is greater than the character string CAT, then the program is to branch to statement 300; otherwise, program execution is to continue with the next sequential statement.

THE COMPUTED GOTO STATEMENT

The computed GOTO statement is a version of the GOTO statement that gives you the ability to branch to different statements during various stages in a program.

A computed GOTO could look like this:

```
0100 GOTO 30,40,50 ON J
```

A branch is made to statement 30, to statement 40, or to statement 50, based on the integer portion of the value contained in the variable J. The integer portion may contain a value of from 1 to 3. If the value is 1, a branch is made to the first statement shown in the list, statement number 30. If the value is 2, the branch is to be the second statement, number 40. If the value is 3, the branch is to the third statement, number 50. If the value is greater than or equal to (≥) 4 or less than (<) 1, program execution *falls through* to the statement following the computed GOTO statement.

The expression determining the branch to be made can be a simple variable, such as J above, or a more complicated expression, say (A + B) / 2. If such an expression were used, its computed value would determine the branch to be made. Consider this example:

```
0050 GOTO 200,220,100,240 ON (A+B)/2
```

The expression (A + B) / 2 is evaluated, and a branch is made to statement number 200, 220, 100, or 240, depending on whether the value is 1, 2, 3, or 4, respectively. Note also that the statement numbers shown in the list do not have to be specified in sequential order; that is, statement number 100 can be the third number in the list even though it is a lower number than the others.

More about Loops–Using FOR and NEXT Statement

A still more concise method of specifying a loop is by using the FOR and NEXT statements. For example, our program for finding and displaying the finance charge for $100 to $5000 could be further simplified to look like this:

```
0010 FOR P=100 TO 5000 STEP 100
0020 PRINT P,P*.015
0030 NEXT P
```

The FOR statement identifies the beginning of the loop; the NEXT statement identifies the end of it. In between is the statement, or sequence of statements (we need only one for this example) that will be executed repeatedly until the specification in the FOR statement has been satisfied.

In our example, the FOR statement specifies that the statement in the loop (the PRINT statement) will be executed repeatedly for successive values of P from 100 through 5000. (An increment of 100 is added to P for each execution of the NEXT statement.) When the value of P exceeds 5000, execution of the loop is ended, and control is passed to the next logically executable statement following the NEXT statement. In this case, the following statement is a STOP statement denoting the end of the program. However, other statements could precede it, or the NEXT could be the last statement prior to the STOP.

The increment is always 1 unless it is explicitly stated to be otherwise; for example:

```
0010 FOR P=100 TO 5000 STEP 200
```

This FOR statement explicitly states an increment (or step) of 200. Thus, the statement(s) in the loop will be executed once for every odd multiple of P from 100 to 5000 (that is, the range is 100, 300, 500,...4900). When the value of P exceeds 5000 (that is, when it reaches 5100), execution of the loop will end. The value of P will be set back to 4900 before the next logically executable statement is executed.

If you want to execute the loop once for every even multiple of 100 to 5000 (that is 200, 400, 600,...5000), you would say the following:

```
0010 FOR P=200 TO 5000 STEP 200
```

Again, when the value of P exceeds 5000 (in this case, when it reaches 5200), execution of the loop will end. The value P will be set back to 5000 when the next logically executable statement is executed.

As with expressions appearing in assignment statements and in the body of PRINT statements, the specifications in FOR statements can be quite complicated. For example, the following FOR statements are permitted:

```
0030 FOR I=A TO B
0040 FOR J=8*M+Y TO A↑3
0050 FOR K=SQR(B)-C TO 550 STEP A/B↑2
```

The first example states that the initial value of I is to be taken from the variable A and that the loop is to be executed repeatedly until the value exceeds the value of B. The second example states that the initial value of J is the value of the expression 8 * M + Y, and the loop is to be executed until this value exceeds the value of A ** 3. The third example states that the initial value of K is to be the square root of B minus C; the loop is to be executed until the value 550 is exceeded, and each time through the loop the value of K is to be increased by the value of the expression A / B↑2.

You can also use more than one set of FOR/NEXT statements together in a program by nesting one loop. Let's look at a program that computes compound interest and uses nested FOR loops in the process.

The mathematical formula to compute compound interest is:

$$A = P\left(1 + \frac{R}{100}\right)^{t}$$

where A is the amount to be calculated, P is the principal, R is the rate of interest, and T is the time period.

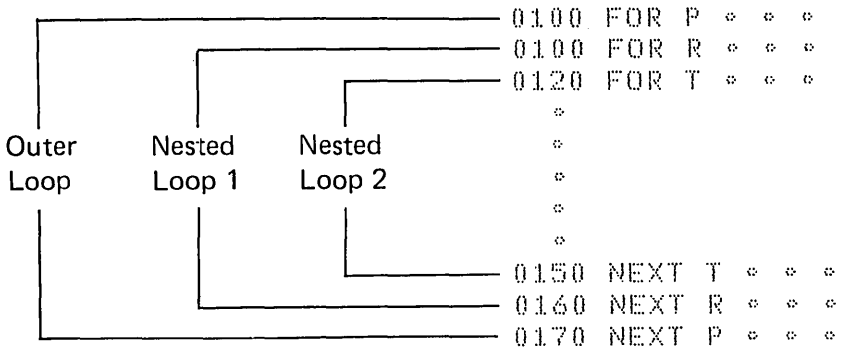
The program below shows how you can enter any amount as the principal (P), compute interest on it using interest rates from 1% to 20%, for each of 10 years, and display all the amounts–a total of 200 values.

```
0090 PRINT 'ENTER PRINCIPAL'
0100 INPUT P
0105 PRINT 'TIME','RATE','AMOUNT'
0110 FOR T=1 TO 10
0120 FOR R=1 TO 20
0130 A=P*(1+R/100)↑T
0140 PRINT T,R,A
0150 NEXT R
0160 NEXT T
```

Statement 130 duplicates, in BASIC terms, the compound interest formula. The FOR statement numbered 120 and the NEXT statement numbered 150 delimit one loop. The first time through the loop, the value of R, the rate variable, is set to 1. When NEXT R is reached, R is incremented by 1 and the statements are executed again with the new value of R. Each time through the loop the PRINT statement prints time, rate, and amount values. This process continues until R reaches 20 and the loop is ended.

However, this loop is enclosed, or *nested*, within the loop delimited by the FOR and NEXT statements numbered 110 and 160. This outer loop changes the value of T, the time variable, from 1 to 10. Each time the value of T changes, the inner loop cycles through 20 times changing the value of R. Since T changes value 10 times, the loop changing the value of R is executed 200 times. Each time, the PRINT statement prints new values.

A nested loop is one that is enclosed by another loop. That is, the FOR/NEXT statements of one loop occur between the FOR/NEXT statements of another loop, as illustrated:



FUNCTIONS AND SUBROUTINES

As part of the BASIC language, you can define functions or write a program segment, called a subroutine, which you expect to use several times in your program.

Functions

User-written functions can be arithmetic or character. An arithmetic function is named by the letters FN followed by a single letter. A character function is named by the letters FN followed by a single letter and the currency symbol ($).

The following can be names of arithmetic functions:

FNA
FNB
FNR
FN#

The following can be names of character functions:

FNA$
FN#$

A user-written function is named and defined by the DEF statement. For example:

```
0010 DEF FNE(X)=EXP(X↑2)
```

defines the natural exponential of X squared, using the intrinsic function EXP. The arithmetic variable X, enclosed in parentheses after the function name FNE, is called a *dummy variable*. You can have more than one dummy variable, and the list of variables can contain both arithmetic and character dummy variables. (The expression value substituted for each dummy variable is called an *argument*.) After defining a function, the function name and its accompanying argument(s) can be used anywhere in your program. For example:

```
0010 DEF FNE (X) = EXP (X↑2)
   .
   .
   .
0050 Y = FNE (.5)
0060 Z = FNE (C+2)
0070 PRINT FNE (3.75)+Y/Z
```

User-defined functions can be defined in one statement or over a group of statements. A function defined in one statement, such as the function illustrated above, is called a *single-line* function. A function defined over many statements is called a *multiline* function. A multiline function begins with the word DEF, the function name, and any arguments, the same as single-line functions. However, the DEF statement does not contain the equal sign or an expression. Rather, the value of the function is developed by the statements following the DEF and is defined in a RETURN statement, which computes the value and returns the value to the program. The end of a multiline function is defined by the FNEND statement. Here is the way the statements in a multiline function must be sequenced:

DEF *function name* [(variables, if any)]
.
.
.
RETURN *expression*
.
.
.
FNEND

Here is an example of a multiline function:

```
0030 DEF FNA (X,Y)
0040 IF X>0 & Y>1 GOTO 60
0050 GOTO 65
0060 RETURN X+Y
0065 RETURN X-Y
0070 FNEND
```

This function uses two dummy arithmetic variables (X and Y) as arguments. The function tests the values of both arguments. If X is greater than 0, and Y is greater than 1, the values are added and the sum is returned to the program. If the values do not satisfy the tested conditions, program control transfers to statement 65. If this function were used in the following program, C would have a value of 7 and D would have a value of -2.

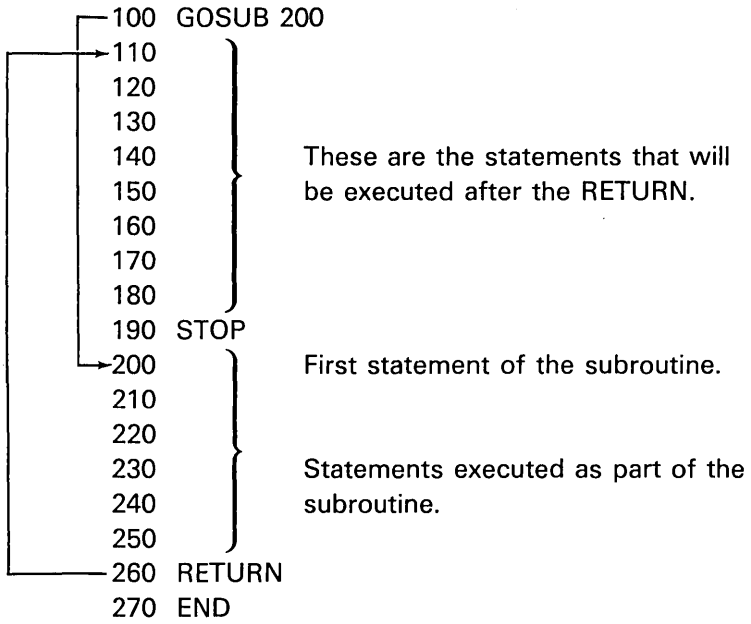
```
0030 DEF FNA (X,Y)
0040 IF X>0 & Y>1 GOTO 60
0050 GOTO 65
0060 RETURN X+Y
0065 RETURN X-Y
0070 FNEND
  .
  .
  .
0100 A=5
0120 B=2
0130 C=FNA (A,B)
0140 D=FNA (0,2)
```

Subroutines

Another way of writing a group of statements to be executed at different times in your program is to group them into a subroutine. Execution of a subroutine begins with the GOSUB statement, where the number specified in the statement specifies the number of the first statement in the subroutine. For example:

```
100    GOSUB 200
```

causes the computer to skip, or *branch*, to statement 200, the first statement in the subroutine. Program execution continues from that point. To cause the computer to branch back to statement 100 (actually, to the next sequential statement following statement 100), the last statement of the subroutine must be a RETURN statement. This RETURN statement, unlike a RETURN used with a function, contains no expression. A program containing a subroutine could be sequenced like this:



Statement 100 branches to statement 200. Statement 260 returns control to statement 110. Statement 190 tells the computer the end of the program has been reached. The STOP statement is similar to an END statement except that higher-numbered statements may follow it. Its use is to denote the end of program execution when the logical conclusion of the program occurs somewhere in the middle of the program, as shown here. The STOP statement here is equivalent to writing GOTO 270.

A program illustrating the use of a subroutine is shown below. This program determines the greatest common divisor of three integers. The first two numbers are selected in program statements 30 and 40, and their greatest common divisor (CD) is determined in the subroutine, statements 200 through 310. The CD just found is assigned to X in statement 60. The third number read in from the INPUT statement is assigned to Y in statement 70. The subroutine is entered a second time from statement 80 to find the greatest common divisor (CD) of these two numbers. The result is, of course, the greatest common divisor of the three given numbers. It is displayed with them in statement 90.

```
0010 PRINT 'ENTER THREE INTEGERS'
0020 INPUT A,B,C
0030 X=A
0040 Y=B
0050 GOSUB 200
0060 X=G
0070 Y=C
0080 GOSUB 200
0090 PRINT 'A','B','C','CD'
0095 PRINT A,B,C,G
0100 STOP
0200 Q=INT(X/Y)
0210 R=X-Q*Y
0220 IF R=0 GOTO 300
0230 X=Y
0240 Y=R
0250 GOTO 200
0300 G=Y
0310 RETURN
```

Let's assume these numbers are entered when the INPUT statement is executed:

```
ENTER THREE INTEGERS

?
60,90,120
```

The output will be:

```
1           19          37          55          (print positions)

A           B           C           CD
 60          90          120         30
```

Another example of input and resulting output is:

```
ENTER THREE INTEGERS

?
32,384,72

A           B           C           CD
 32          384         72          8
```

COMPUTED GOSUB STATEMENT

The computed GOSUB statement is similar to the computed GOTO statement discussed in this chapter. They both cause a branch to one of a number of statements based on the computed value of an expression. The difference between the two statements is that the GOSUB branches to a subroutine; the RETURN statement in the subroutine returns program execution to the statement following the computed GOSUB statement.

Consider this example:

```
0030 GOSUB 120,175,195 ON X-Y
```

A branch is made to one of three subroutines, either the one beginning with statement 120, the one beginning with statement 175, or the one beginning with statement 195, depending on whether the integer portion of the value contained in the expression X - Y is 1, 2, or 3, respectively. If the expression X - Y results in a value other than 1, 2, or 3, program execution continues with the statement following the GOSUB.

PROGRAM CHAINING

With the program chaining technique, a BASIC program can be shared with other BASIC programs. For example, suppose that when writing a program you discover that an operation you want to perform is available as a separate program. It could be time saving to you to be able to use that program in conjunction with the one you are currently writing. The CHAIN and USE statements can help you access data and execute that program.

The CHAIN statement is used in one BASIC program to tell the computer to stop executing the current program and start executing another BASIC program. To tell the computer which program to start executing, you name it in the CHAIN statement. Here's an example:

```
0500 CHAIN 'D40','PROGB'
```

This statement instructs the computer to begin executing the program (in diskette drive 2) named PROGB. Note that when the CHAIN statement is executed, the current program (the program containing the CHAIN statement) is terminated.

Variable values in the chaining program are passed to the chained program; that is, they become accessible for use in that program only if they were defined in a USE statement.

In the program being chained, the USE statement specifies a list of variables that will receive the values passed from the CHAIN statement. For example, the value passed by J$ to PROGB can be received by PROGB in the statement:

```
0200 USE K$18
```

Note that the USE statement is written in both programs and the CHAIN statement is written in the chaining program (the program requesting execution of another program). The USE statement must be the first statement referencing a variable in each program.

The CHAIN and USE statements derive their value in being able to help you string two or more programs together instead of having to code similar program sections for individual programs. Also, CHAIN and USE statements allow you to segment large programs. The following is an example of CHAIN and USE.

```
0010 REM THIS IS PROGA
0020 USE J$
  .
  .
  .
0300 CHAIN 'D40','PROGB'
```

Chaining program

```
0010 REM THIS IS PROGB
  .
  .
  .
0050 USE K$
  .
  .
```

Program being chained

The CHAIN statement at 0300 of PROGA causes PROGB to be loaded into storage, then execution transfers to PROGB. The value in J$ of PROGA is passed to K$ as specified by the USE statements in both programs.

The PRINT USING FLP statement is useful for controlling the format of a report. PRINT USING FLP is used in conjunction with an Image or FORM statement or character variable to print values according to the format specified by the statement or variable. The PRINT USING FLP statement includes the values to be printed and the statement number or character variable of a corresponding Image or FORM statement that specifies the format of the print line. For example:

```
0030 PRINT USING 40,FLP,N,A
```

This statement refers to statement number 40, an Image or FORM statement, which will cause the computer to format the arithmetic variables N and A on the print line.

PRINT USING AND THE IMAGE STATEMENT

Statement 40 could look like this:

```
0040 : IN ## YRS AMT = $#####.##
```

The colon beginning statement 40 identifies it as an Image statement. The alphabetic characters are printed exactly as they appear in the statement, and the pound sign (#) is the symbol used to indicate that a value will be supplied from the output list in the PRINT USING statement. The value of N replaces the first set of #'s, and the value of A replaces the final set. The decimal point in the final set indicates that the value of A is to be aligned on the decimal point in the image specification.

If N contains the value 10 and A contains the value 1628.88, the output line produced by statements 30 and 40 would look like:

```
IN 10 YRS AMT = $1628.88
```

In the Image statement, the pound sign (#) is used as a placeholder. In statement 40, the first set of #'s indicates that a value is to be displayed using two positions; the second set displays a value over six positions aligned on a decimal point between the fourth and fifth positions. If the value to be printed is smaller than six digits (say the value 300.40), the first, or *high-order* position, would be printed with a blank. Excess decimal positions are rounded to the number of decimal position # signs.

The Image statement can also contain a placeholder consisting of the symbols |||| for an exponential value. If you want to print a value containing an exponent, the Image statement could contain the following sequence of symbols:

```
##.##↑↑↑↑
##.##||||
```

This sequence states that a value is to be printed with four digits followed by an exponential value. An exponential value is always printed with four positions for the format: E±dd. The letter E is followed by a plus or minus sign indicating a positive or negative exponent, followed by two digits. Therefore, a set of four |s must always be specified as placeholders for exponents. If an Image specification contains this sequence:

```
##.##||||
```

the following shows how different values would be printed by that sequence:

Value	Printed Format
123	12.30E+01
12.3	12.30E+00
.123	12.30E-02

The specification calls for four digits to be printed aligned on the decimal point. Therefore, the number 123 is represented as 12.30 with an exponent of +1. The exponent tells us two things: the direction in which the decimal point is to be moved (+, to the right, and -, to the left), and the number of digits over which it is to be moved. In the first example, the exponent +1 tells us to move the decimal point one position to the right; the number 12.30E+01 is the same as 123. In the second example, the number 12.3 can be aligned on the decimal point with no action required by the exponent, hence an exponent of E+00; the number 12.30E+00 is the same as 12.3. The third example tells us to move the decimal point two positions to the left; 12.30E-02 is the same as .123.

Blank positions in an array referenced by a PRINT USING statement are significant. The entered characters of a variable do not determine the size of the variable to be used by the PRINT USING statement. For example, with a variable A$ dimensioned to 30 for the entry of a name, and the name C. A. JONES entered into the variable, the PRINT USING statement will use all 30 positions of the variable, including the blank positions.

Note that a PRINT USING statement can be ended with a semicolon (;) to suppress printing of a new line and cause subsequent printing to occur on the same line, as shown in the following example.

Example:

```
0010 PRINT 'ENTER TODAY''S DATE'
0020 INPUT D$
0030 PRINT USING 40,FLP,D$
0040 :MONTHLY SALES BY SALESMAN AS OF ##########
0050 PRINT FLP
0060 PRINT FLP
0070 PRINT USING 100,FLP
0080 PRINT USING 110,FLP,;
0090 PRINT USING 120,FLP
0100 :SALESMAN            SALESMAN
0110 :NAME                NUMBER        GROSS SALES        EXPENSES
0120 :        NET SALES
0130 PRINT FLP
0140 PRINT 'ENTER SALESMAN''S NAME'
0150 PRINT 'OR ENTER  END  TO END'
0160 INPUT A$
0170 IF A$='END' GOTO 300
0180 PRINT 'ENTER SALESMAN''S NUMBER'
0190 INPUT D
0200 PRINT 'ENTER GROSS SALES'
0210 INPUT A
0220 PRINT 'ENTER EXPENSES'
0230 INPUT B
0240 C=A-B
0250 PRINT USING 270,FLP,A$,D,A,B;
0260 PRINT USING 280,FLP,C
0270 :##################   ###            $#####.##            $####.##
0280 :         $#####.##
0290 GOTO 140
0300 STOP
```

In this sample program, statements 40, 100, 110, 120, 270, and 280 are Image statements used to format the printed report shown below.

```
MONTHLY SALES BY SALESMAN AS OF 10/19/77

SALESMAN        SALESMAN
NAME            NUMBER     GROSS SALES   EXPENSES   NET SALES

R. DOVER        623        $5678.00      $352.00    $5326.00
G. FREDERICK    571        $2534.00      $152.00    $2382.00
A. JOHNSON      860        $8564.00      $561.00    $8003.00
D. SMITH        487        $5346.00      $356.00    $4990.00
```

PRINT USING AND THE FORM STATEMENT

The FORM statement, offers greater formatting capabilities than the Image statement. For example, it provides a special code to specify character data. It contains format control specifications to tell the computer how to position output on a print line; one of these specifications, SKIP, must be coded on the FORM statement to cause a line to be printed.

Numeric Specification–PIC

The PIC specification in the FORM statement shows a *picture* of the way a number should be formatted. This picture is enclosed in parentheses. The symbols #, ., and |, previously illustrated in the Image statement, could be used in the FORM statement in this format:

```
PIC(##.##|||)
```

You recall that the # symbol is used as a placeholder for a digit and the I symbol is used as a placeholder for an exponent. The PIC specification has these additional placeholders, or *digit specifiers*:

Symbol	Meaning
Z	A leading zero is to be replaced with a blank.
*	A leading zero is to be replaced with an asterisk.
$	Floating dollar sign. A dollar sign is to be printed immediately before the first significant digit.
+	Floating plus sign. A plus sign for a positive number, or a minus sign for a negative number, is to be printed immediately before the first significant digit.
-	Floating minus sign. A minus sign for a negative number, or a blank for a positive number, is to be printed immediately before the first significant digit.

Here are examples of digit specifiers. Assume that a data item containing the value 112233 is to be printed.

PIC Specification	Printed Output
PIC(##########)	00112233
PIC(ZZZZZZZZZ)	112233
PIC(ZZZZZZ###)	112233
PIC(*******##)	***112233
PIC($$$$$$$###)	$112233
PIC(+++++++###)	+112233
PIC(---########)	112233

If a floating character (dollar sign, plus sign, or minus sign) is specified only once at the start of a PIC specification, it does not float through the field but instead is printed in the indicated position. For example:

PIC Specification	Printed Output
PIC($ZZZZZ###)	$ 112233
PIC(+ZZZZ###)	+ 112233

The PIC specification can also contain *insertion characters* to edit a printed item. Digit specifiers indicate how the number itself is to be treated; insertion characters simply insert additional characters into a field, generally to improve readability. The following insertion characters can be specified:

Symbol	Meaning
B	Print a blank unconditionally.
,	Print a comma conditionally (only if a digit precedes the comma).
/	Print a slash conditionally (only if a digit precedes the slash).
.	Print a decimal point conditionally (if the value to be printed is nonzero or zero suppression (Z) is not in effect).
+	Trailing plus sign. When the + appears in the rightmost position of a PIC specification, it is treated as a trailing sign. A plus sign is printed for a positive number, a minus sign for a negative number.
-	Trailing minus sign. When the - appears in the rightmost position of a PIC specification, it is treated as a trailing sign. A minus sign is printed for a negative number, a blank for a positive number.
CR DB	When the characters CR or DB appear at the end of a PIC specification, they are treated as a trailing sign. CR or DB is printed for a negative number; blanks are printed for a positive number.

Here are examples of insertion characters added to the examples previously shown:

PIC Specification	Printed Output
PIC(###B##B#####)	000 11 2233
PIC(ZZZBZZBZ####)	11 2233
PIC(ZZZ,ZZZ,###)	112,233
PIC(ZZZZZ/Z#/##)	11/22/33
PIC(*******#.##)	*112233.00
PIC($$$$$$$###+)	$112233+
PIC($$$,$$$,$$$.##)	$112,233.00

In the first example, a blank is entered after the third and fifth digits. Because # is denoted as the digit specifier, leading zeros are not suppressed.

The second example illustrates the blank used with the Z digit specifier, which does suppress leading zeros.

The third example illustrates the use of commas. The first comma is not printed because no digit precedes it (zero suppression having been specified); the second comma is printed.

The fourth example inserts slashes.

The fifth example illustrates the effect of a decimal point; because the number 112233 is an integer number, it is aligned on the decimal point, and zeros print out in the decimal portion of the field.

The sixth example adds a trailing sign to a field that also contains floating dollar signs.

The last example adds commas and a decimal point to format a dollar amount. Note that the first comma is not printed, but its absence is marked by a blank, as it was in the third example. The dollar sign floats over the comma.

Character Specification–C

Unlike the Image statement, the FORM statement specifies a place where character data is to appear, indicated by the specification code C. The actual character data is written in the PRINT USING statement.

To print both character and numeric data, a PRINT USING statement could be written like this:

```
0030 PRINT USING 50,FLP,'COST OF',A1,'CHAIRS IS',B1
```

The corresponding FORM statement could look like this:

```
0050 FORM C,PIC(Z#),C,PIC($$$,$$#.##)
```

The first appearance of the letter C indicates that a character string from the PRINT statement is to be printed. The first PIC specification describes the arithmetic variable A1; if the value is zero, a blank is printed in the leftmost position, followed by a zero. The second C describes the second character string, and the second PIC describes the variable B1.

The C specification code marks a place for character data regardless of the number of characters to be printed. You could specify the exact number of characters to be printed by indicating the number after the C code. For example:

```
C6
```

This specification indicates that 6 characters are to be printed. Care should be used when specifying a number because only that number of characters is printed. For example, if you specify C6 to print the character string COST OF, only the characters COST O will be printed.

Format Control Specifications–X, POS, SKIP

Format control specifications provide flexibility in formatting an output line. These specifications allow you to space over a number of print positions on a line, to specify the print position where a data item is to begin printing, and to skip print lines.

The Xn specification spaces over *n* print positions. For example, X10 causes the printer to space the next 10 positions before printing a data item.

The POSn specification prints a data item beginning in position *n*. For example, POS50 causes the next data item to print beginning in position 50.

The SKIPn specification skips *n* print lines. To skip five lines, specify SKIP5. To skip to the next line, specify SKIP1 or SKIP with no number. For example, to cause statement 50, shown earlier, to print a line, SKIP must be added to it:

```
0050 FORM C,PIC(Z#),C,PIC($$$,$$#.##),SKIP
```

Statement 50 is now complete, and if combined with PRINT USING FLP statement number 30,

```
0030 PRINT USING 50,FLP,'COST OF ',A1,'CHAIRS IS',B1
```

results in this output:

```
COST OF 14 CHAIRS IS $1,510.00
```

Here are additional statements using format control specifications:

Example 1:

```
0140 PRINT USING 145,FLP,A1,B1
0145 FORM POS15,PIC(Z#),POS32,PIC($$$,$$#.##),SKIP1
```

Statement 145 uses the POS and SKIP control specifications. POS15 positions the printer at position 15 before printing the value contained in A1 described by the PIC specification. POS32 begins printing the value of B1 at position 32. After all printing is complete, SKIP1 causes the carriage to skip to the next line.

Example 2:

```
0110 PRINT USING 115,FLP,'COST OF ',A1,'CHAIRS IS',B1
0115 FORM X5,C,POS15,PIC(Z#),POS20,C,POS32,PIC($$$,$$#.##),SKIP1
```

In statement 115, X5 states that the first five positions of the print line are to be skipped, and the character data controlled by the C code, the string COST OF, is to be printed. POS15 prints the value of A1 beginning in position 15. POS20 prints the character string CHAIRS IS beginning in position 20. POS32 prints the value of B1 beginning in position 32. SKIP1 causes the line to be printed.

Following is a program that uses these statements.

```
0100 A1=15
0105 B1=A1*115.25
0110 PRINT USING 40,FLP,'COST OF ',A1,'CHAIRS IS',B1
0115 FORM X5,C,POS15,PIC(Z#),POS20,C,POS32,PIC($$$,$$#.##),SKIP1
0120 FOR A1=14 TO 1 STEP -1
0130 B1=A1*115.25
0140 PRINT USING 145,FLP,A1,B1
0145 FORM POS15,PIC(Z#),POS32,PIC($$$,$$#.##),SKIP1
0150 NEXT A1
```

This program finds the cost of 1 to 15 chairs at $115.25 each. Statements 110 and 115 print out the first line, statements 140 and 145 print out all succeeding lines based on the loop defined between statements 120 and 150.

Output from this program will look like this:

Print Position	**6**	**15**	**20**	**32**
	COST OF	15	CHAIRS IS	$1,728.75
		14		$1,613.50
		13		$1,498.25
		12		$1,383.00
		11		$1,267.75
		10		$1,152.50
		9		$1,037.25
		8		$922.00
		7		$806.75
		6		$691.50
		5		$576.25
		4		$461.00
		3		$345.75
		2		$230.50
		1		$115.25

Example 3:

```
0050 R$='WINS'
0060 IF A>B GOTO 80
0065 R$='LOSES'
0070 IF A≠B GOTO 80
0075 R$='TIES'
0080 PRINT USING 90, FLP,HOME TEAM XXXXXX FINAL SCORE A,'-',B,R$
0090 FORM POS10,C,PIC(Z#),C,PIC(Z#),POS20,C6,SKIP
```

Statement 90 uses the POS20 and C6 control specifications to overlay position 20 of the print line with the value of R$. If A is 3 and B is 24, the printed line will look like this:

Print Position **10** **20** **28**

```
HOME TEAM LOSES  FINAL SCORE 3-24
```

PRINT USING WITH A CHARACTER VARIABLE

In addition to the Image and FORM statements for output formatting, BASIC also allows assignment of a format to a character variable which can then be referenced in input/output statements. Each character variable to be used in this manner should first be dimensioned (in a DIM statement) to the length of the format. The format assigned to the character variable is identical to the format following the colon in an Image statement, or the first 4 characters can be FORM, followed by the format specifications normally entered for a FORM statement. The following example shows the use of character variables for formatting.

```
0010 DIM A$50,B$100,C$100,D$100,E$100
0020 A$='MONTHLY SALES BY SALESMAN AS OF ##########'
0030 PRINT 'ENTER DATE'
0040 INPUT D$
0050 PRINT USING A$,FLP,D$
0060 PRINT FLP
0070 PRINT FLP
0080 B$='SALESMAN    SALESMAN      GROSS      EXPENSES     NET'
0090 C$='NAME        NUMBER        SALES                     SALES'
0095 PRINT FLP,B$
0096 PRINT FLP,C$
0100 PRINT FLP
0110 PRINT 'ENTER SALESMAN''S NAME OR ENTER STOP TO END'
0130 INPUT N$
0140 IF N$='STOP' GOTO 280
0150 PRINT 'ENTER SALESMAN''S NUMBER'
0160 INPUT S
0170 PRINT 'ENTER GROSS SALES'
0180 INPUT G
0190 PRINT 'ENTER EXPENSES'
0200 INPUT E
0210 T=G-E
0220 D$='############ ###          $#####.##  $#####.##    $#####.##'
0230 PRINT USING D$,FLP,N$,S,G,E,T
0240 PRINT FLP
0250 GOTO 110
0280 STOP
```

Printer Spacing Control

You can use the contents of file FLS to control the number of lines printed per inch (see Chapter 12 for more information about file FLS). The printer normally prints 6 lines per inch, with 16 increments of the print roll per line, for a total of 96 increments. You can change the number of lines per inch by entering a number between 8 and 99 in the tenth and eleventh positions of file FLS. The number you enter is divided into 96 to determine the number of lines per inch. For example, if you enter:

8	– 12	lines are printed per inch
12	– 8	lines are printed per inch
16	– 6	lines are printed per inch
24	– 4	lines are printed per inch
32	– 3	lines are printed per inch

An entry of less than 12 will cause printing to be overlapped. An entry of zero causes suppression of spacing, which results in lines printed right over preceding lines. Sample WRITE FILE statements for printer spacing control are shown below:

```
0010 WRITEFILE FLS,'          96'
0020 PRINT FLP,'LINE PRINTED 1 PER INCH'
0030 WRITEFILE FLS,'          16'
0040 FOR I=1 TO 6
0050 PRINT FLP,'6 LINES PER INCH'
0060 NEXT I
0070 WRITEFILE FLS,'          12'
0080 FOR I=1 TO 8
0090 PRINT FLP,'8 LINES PER INCH'
0100 NEXT I
0110 WRITEFILE FLS,'          00'
0120 PRINT FLP,'NO SPACE TO OVERPRINT'
0130 PRINT FLP,'              OVERPRINT'
0140 PRINT FLP,'              OVERPRINT'
```

```
LINE PRINTED 1 PER INCH

6 LINES PER INCH
6 LINES PER INCH
6 LINES PER INCH
6 LINES PER INCH
6 LINES PER INCH
6 LINES PER INCH
8 LINES PER INCH
8 LINES PER INCH
8 LINES PER INCH
8 LINES PER INCH
8 LINES PER INCH
8 LINES PER INCH
8 LINES PER INCH
NO SPACE TO OVERPRINT
```

Chapter 5. Saving and Loading the Work Area

In this chapter, the following topics concerning saving and loading the work area are discussed:

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

DETERMINING THE SIZE A FILE SHOULD BE

Before information can be stored on tape or diskette, the files must be formatted by the MARK command. When you use the MARK command, you can determine the size of a saved work area by comparing the amount of work area available before and after you have entered data or programs into the work area; therefore, the file size equals the storage available before entering data or programs minus the current storage available divided by 1024.

Saving and Loading Data on a Tape or Diskette File

You can write (save) the contents of a work area to tape or diskette using the SAVE command. This allows you to enter data or programs into the 5110 work area and save this information for later use. Individual data records can also be written to a data file.

Once the contents of the work area are saved in a tape or diskette file, that information can be read back into the work area using the LOAD command. This allows you to load and execute the same program any number of times. You can use a CHAIN statement in a program to end that program and load and execute another program that is saved on tape or diskette.

Controlling the Files on Tape or Diskette

Once you have stored several work areas and data files on a tape or diskette, you might want to know what files you have in your library (stored on tape or diskette). You can use the UTIL command to display a directory of file information for a specified tape or diskette. The directory provides you with such information as the file number, the file ID, and the file type. See the *UTIL Command* in the *IBM 5110 BASIC Reference Manual*, SA21-9308, for a complete description of the information contained in the file directory.

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

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

Maintaining Data Security

You should protect your programs and data from unauthorized access or accidental destruction. Several functions are built into the 5110 to assist you in protecting your programs and sensitive data.

Protecting Your Programs

After you have developed a program, you might want to keep a duplicate (backup) copy of the program on another diskette or tape. Then if the original program is accidentally destroyed, you still have the backup copy available. See the *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311, for information on copying programs. You can use the SAVE command to lock a program so that it cannot be listed or modified. For example:

```
SAVE 5,'MASTER',LOCK,D80
```

This command writes the program to file 5, diskette drive 1, and permanently locks the program against listing or modifying. However, the program can still be loaded and run:

```
LOAD 5,'MASTER',D80
```

When you lock a program, you should also keep an unlocked master copy available in a secure area. This unlocked copy can then be used if the program must be modified.

Protecting Your Data Files

Following are several ways to protect your data files:

- Maintain backup data files
- Use the file write-protect
- Use the diskette access-protect
- Use the SAFE switch on the tape cartridge

Maintaining Backup Data Files

You should keep a backup copy of your data files on another diskette or tape. Then if you accidentally destroy a data file or you encounter a faulty diskette, you can recover your data with minimum effort. To create a backup data file, you periodically copy the master data file onto another tape or diskette.

Using File Write Protection

Accidentally writing to the wrong data file can be prevented by using file write-protection. You can write-protect a file, preventing data from being written to the file, with the UTIL command. For example:

```
UTIL PROTECT 'MASTER',D80
```

Once the file is write-protected, data cannot be written to the file using the SAVE or WRITE commands. However, you can still use the REWRITE FILE statement to update records in the file. To turn off the file write-protection so that you can write data to the file, use the UTIL command. For example:

```
UTIL PROTECT OFF 'MASTER',D80
```

Using Diskette Access Protection

You can use the diskette volume ID, owner ID, and access protection to prevent the wrong diskette from being used for an application. For example, suppose you have a master diskette for an accounts payable application. After you have updated the accounts payable master data files, you could use the UTIL command to specify the volume ID, owner ID, and access protection. For example:

```
UTIL VOLID APMAST,CLARK,ON,D80
```

This command protects the diskette with a volume ID of APMAST, owner ID CLARK, on diskette drive 1 from being accessed. To turn off access-protection, you use the UTIL command and exactly match the diskette volume and owner ID. For example:

```
UTIL VOLID APMAST,CLARK,OFF,D80
```

The UTIL command can be used in a procedure file to prevent the wrong diskette from being used in an application (see Chapter 3 for more information on proecedure files). For example, a procedure file might contain the following records:

```
LOAD 3,'AP.DAILY',D40
UTIL VOLID APMAST,CLARK,OFF,D80
RUN

UTIL VOLID APMAST,CLARK,ON,D80
```

The commands in the procedure file do the following operations:

1. Load the application program from diskette drive 2.
2. Turn off access protection if the proper diskette is in diskette drive 1.
3. Executes the application program.
4. Turn on access protection when the application program has completed execution.

If the wrong diskette was in diskette drive 1, an error occurs when the first command is executed, and the application program is not executed.

Removing Sensitive Data

You are responsible for the security of any sensitive data. After you are through using the system, you can remove the data in the work area by one of the following:

- Using the LOADO command to clear the workarea
- Pressing the RESTART switch
- Turning the POWER ON/OFF switch to OFF

Several methods are available for removing sensitive data from a file. These methods are:

- UTILDROP
- Rewriting a file (OPEN for output, then CLOSE), which makes the old data inaccessible.
- Filling a data file with meaningless data. For example, the following set of statements fills file 1 (on the built-in 5110 Model 1 tape unit) with zeros:

```
0010 DIM A(100)
0020 OPEN FL0,'E80',1,OUT
0030 MAT PUT FL0,A,EOF 50
0040 GOTO 30
0050 STOP
```

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

- How to format the tape
- How much storage on a tape cartridge is actually available to you

HOW TO FORMAT THE TAPE

You must use the MARK command to format files on the tape before you can store work area or data records on the tape. For example:

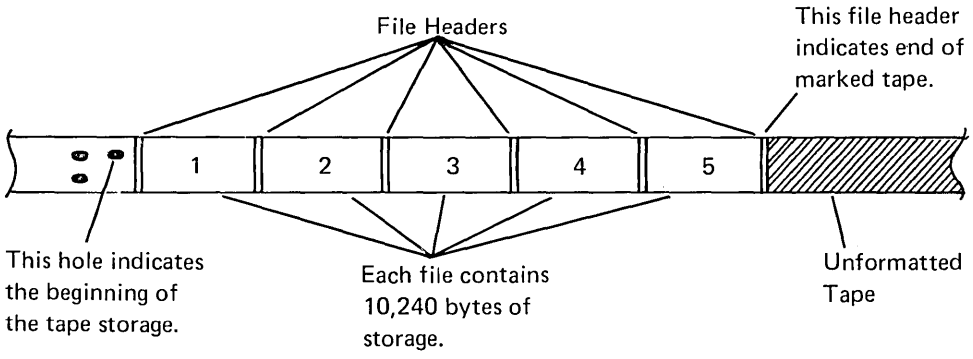
```
MARK    10,     5,      1
```

Starting File Number

Number of Files To Mark

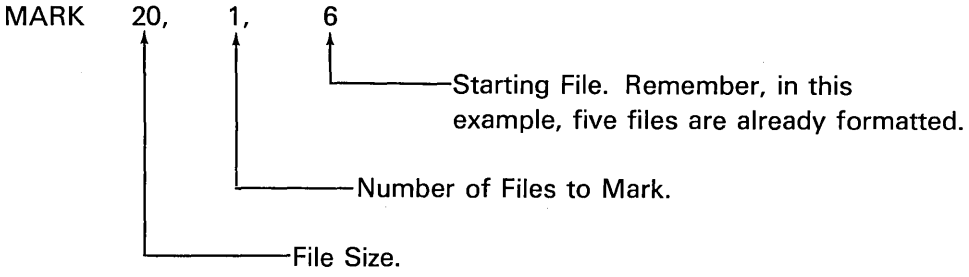
Size of Files in 1,024 (1K) Byte Increments. In this case, the size of the marked files is 10,240 (10 X 1,024)

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

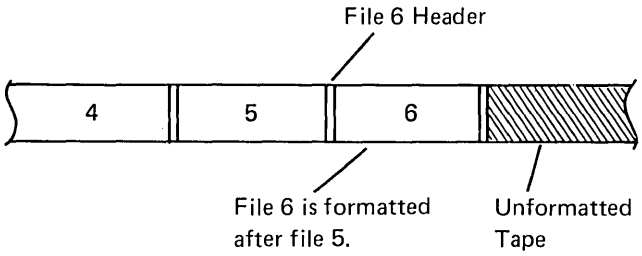


The file headers contain information about the file, such as file number, file name, and file type. Each file header requires 512 bytes of tape storage.

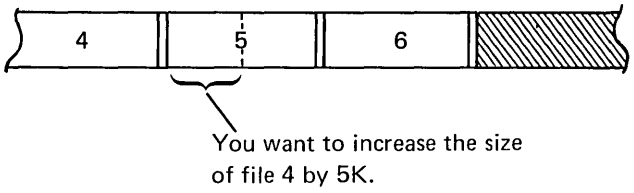
If you want to format additional files on the tape, you must use the MARK command again. For example:



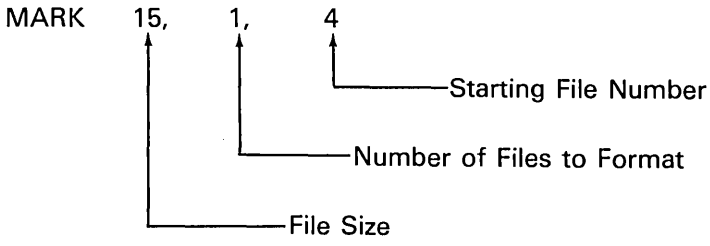
The tape is now formatted as follows:



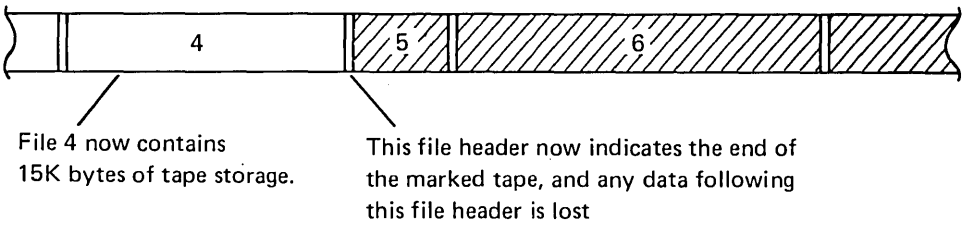
When the information in a tape file is no longer needed, you can use the UTILDROP command to mark the file unused. The defined file space remains available for other uses. However, once a file is formatted, you cannot increase the size of the file without remarking the file. When you remark an existing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



After the command:



is successfully completed, the tape is formatted as follows:



A formatted tape has the following characteristics:

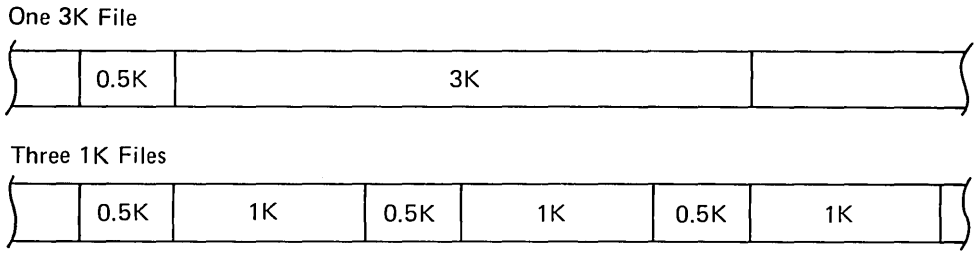
- Files are of variable length from 1K to 204K.
- Files can be randomly accessed; that is, you can read a file without having to read the previous file. Data in the files can be accessed sequentially or randomly.
- Both work area and data files can be on tape.
- Both APL and BASIC files can be on tape.

HOW TO DETERMINE THE STORAGE AVAILABLE ON A TAPE CARTRIDGE

Each tape cartridge contains approximately 204K bytes of storage, but the amount of tape storage actually available to you depends on:

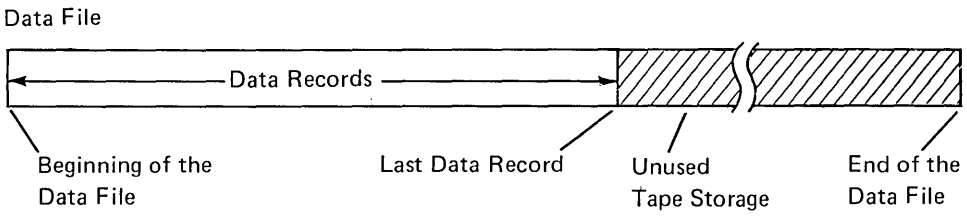
- How many files are marked (formatted) on the tape
- How the data files were written to tape

As mentioned, each file on a tape cartridge requires one 512-byte file header. Therefore, the more files you mark on a tape cartridge, the 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. If you mark three 1K files on tape, however, 1536 bytes of tape storage are required for the three file headers.



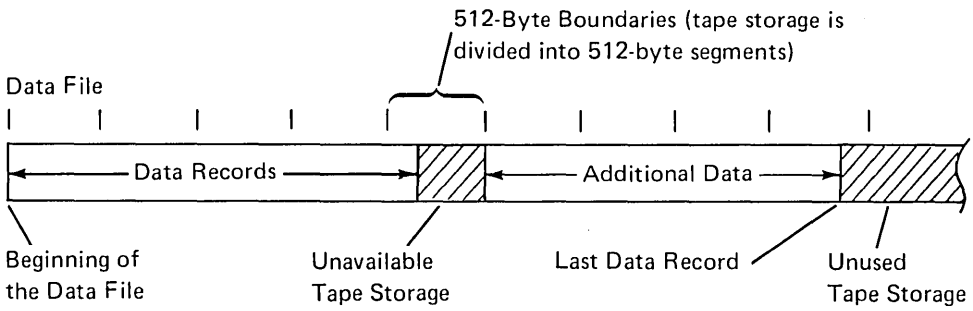
Note, in these examples, that a total of 3K bytes of tape storage is allocated for tape files, although, for the three 1K files, an additional 1K bytes of tape storage are used for headers.

The amount of data you can store in a data file depends on how the data is written to the file. (See Chapters 8, 9, and 10 for a complete description of writing data to the data files.) For example, when you first write data to a data file, 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:



When you add data to the stream I/O data file (see Chapter 8), 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.

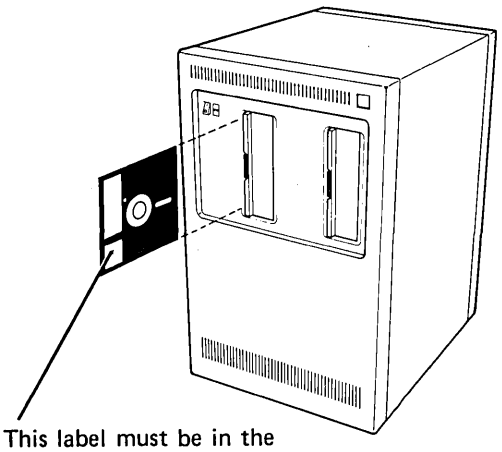
Once the new data records are written to tape, the data file might look like this:



As you add more data to the file, it is possible for more tape storage to become unavailable.

Chapter 7. 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 address locations. 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 system finds the desired address and then reads the data into the 5110.

Before being shipped to a user, each diskette is initialized. Initialization is a process whereby label information and data addresses are recorded on the diskette. Initialization is discussed later in this section.

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 the guidelines in the *IBM 5110 BASIC Reference Manual*.) 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 the development of guidelines for determining when the media should be replaced.

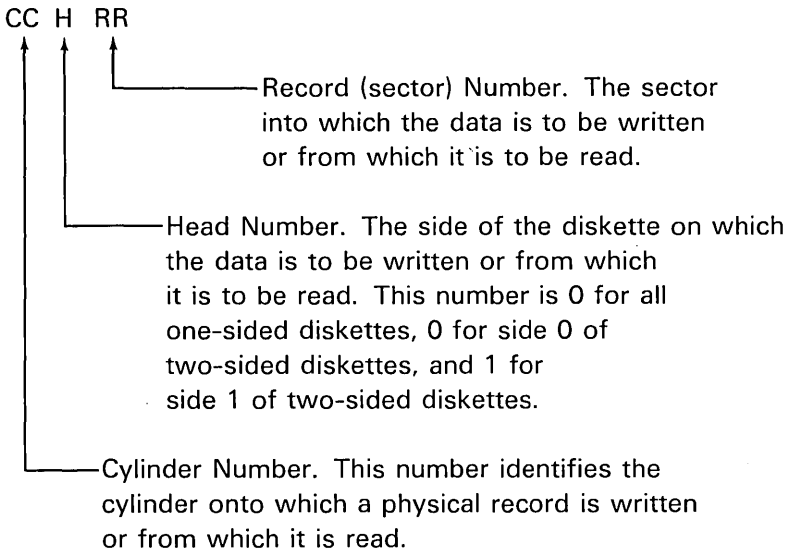
Unpredictable circumstances such as contamination or severe handling can cause an early error to occur.

For all the above 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*).

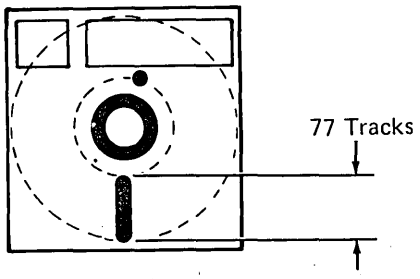
DISKETTE ADDRESSING AND LAYOUT

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 that passes the read/write head while the diskette 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 diskette. Therefore, each diskette has 77 concentric tracks on which data can be stored.

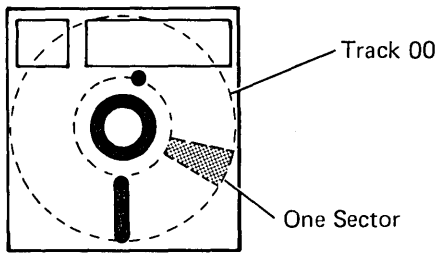


The diskette drive for two-sided diskettes has a read/write head on each side. Each track on side 0 of a two-sided diskette has an associated track on side 1.

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 on a diskette, numbered 0 to 76.

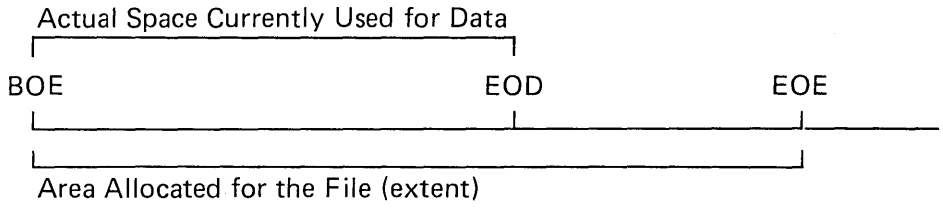
Sector

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



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 might 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 utility 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:

- One-sided, where data is recorded on just one side (Diskette 1)
- Two-sided, where data is recorded on both sides (Diskette 2)
- Two-sided, where data is recorded on both sides at double density (Diskette 2D)

The diskettes are initialized (see *Disk Initialization*) into various formats, consisting of:

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

The 5110 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	1024

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

DISKETTE INITIALIZATION

Diskettes must be initialized before they can be used for storing data. All 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 disk initialization program.

VOLUME ID, OWNER ID, AND VOLUME-PROTECT INDICATOR

Each initialized diskette has a volume ID, owner ID, and volume 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 volume-protect indicator is used to prevent unauthorized access to the diskette volume.

The UTILVOLID command is used to display or change the volume ID and owner ID, or to change the volume-protect indicator.

FILE ID

Each file on a diskette has a file ID (file name). When the diskette files are formatted, a file ID is automatically generated, even though the files are unused. For example, the file ID for file 1 is SYS0001. See the *IBM 5110 BASIC Reference Manual*, for more information on file names when storing data.

DISKETTE FILE WRITE-PROTECT INDICATOR

Each file header contains a write-protect indicator. When the write-protect indicator is *on*, the file can be read into storage and updated, but existing data on the diskette cannot be replaced with new data. The UTILPROTECT command invokes or removes the write-protect indicator for a diskette file.

DISKETTE FILE ORGANIZATION

You must use the MARK command to allocate file space on the diskette before you can store work area or data records on the diskette. For example:

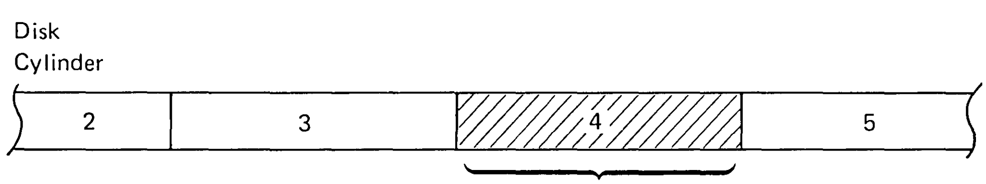
```
MARK   10,    5,     1,     D80
        ↑     ↑      ↑       ↑
        |     |      |       └──── Diskette Drive 1
        |     |      └────────── Starting File Number
        |     └───────────────── Number of Files to Format
        └─────────────────────── Size of the Files in 1024 (1K) Byte Increments.
                                 In this case, the size of the marked files is
                                 10240 (10x1024) bytes.
```

Unlike tape files, 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 UTILDROP command to mark the file unused. Defined space of the file remains available for other uses. However, once a file is formatted, you cannot increase the size of the file without remarking the file. Reallocating diskette file space is discussed next.

REALLOCATING DISKETTE FILE SPACE

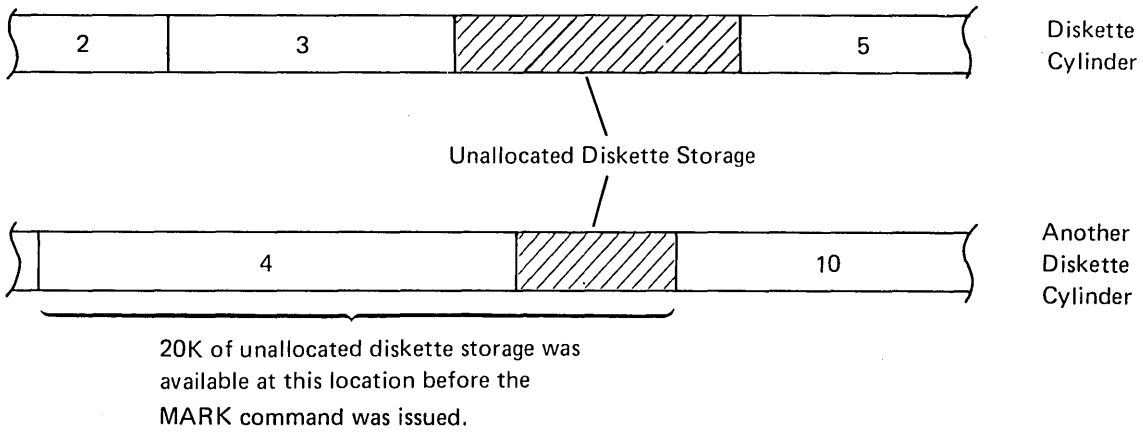
Unlike tape files, when you re-mark an existing diskette file, no other diskette files are affected. When you re-mark a diskette file to increase the size, the file space presently allocated to that diskette file is made available for other files being marked. The re-marked file will then be located 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, D80 command:



After you issue the MARK command this file space is no longer allocated for File 4.

Once the file space previously occupied by file 4 is available, that file space will be used by 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.



DETERMINING THE STORAGE AVAILABLE ON A DISKETTE

Available diskette storage varies, depending upon the type of diskette being used. The amount of storage depends on:

- 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 called the *index track* and is reserved for information (file headers) about the diskette files. Cylinders 75 and 76 are alternate cylinders 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	1024	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 and the size of the files marked on the diskette
- The types of data files written to the diskette
- How the file space is allocated from previous MARK and UTILFREE commands
- Whether an extended label area was requested at initialization time (see *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311)

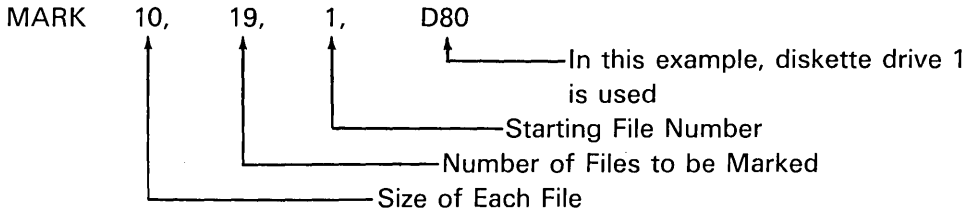
Number and Size of the Diskette Files

Generally, there is a maximum number of files that can be on a diskette:

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

If you use diskette 2D, see the disk initialization program in the *IBM 5110 Customer Support Functions Reference Manual*, 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, and you issue the following command:

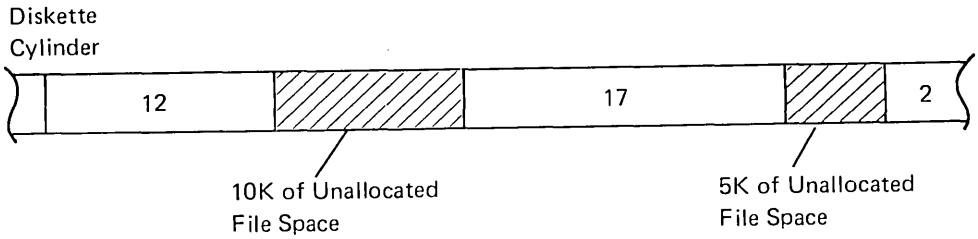


This command marks the diskette with the maximum of 19 files. Each file is 10K bytes; therefore, a maximum of 190K (194,560) bytes of storage is allocated for the files. Now, if you subtract the allocated diskette storage from the available diskette file space:

```
  246,272
- 194,560
  51,712 <-- This much diskette storage is
             unused and unavailable for you
             to store data.
```

How the File Space is Allocated

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

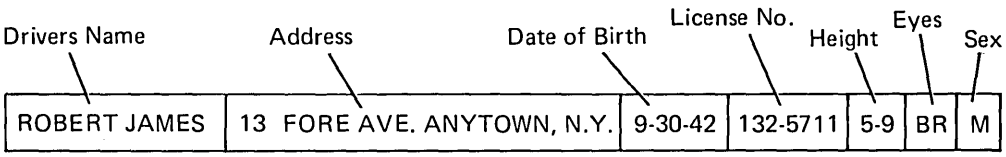


If you need that 15K of storage to mark a new file, the storage is not available because it is not in 15K contiguous bytes.

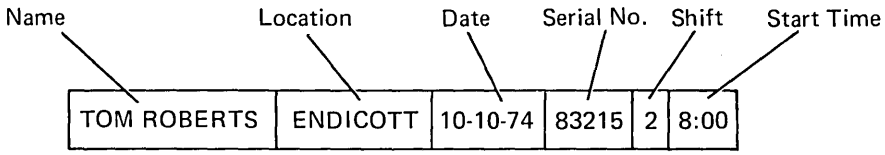
The fragmented blocks of unallocated file space can be made available by the compress function (see the *IBM 5110 Customer Support Functions Reference Manual*). The compress function closes the gaps caused by the unallocated file space and places all of the unallocated file space in one contiguous area.

FILES, RECORDS, AND FIELDS

A file is a collection of related data items which are grouped together in *records*. Most of us carry a driver's license. That is a record. What about a time card? That too is a record. Each of those records contains items related to the purpose of the specific document. The related items are called *fields*. The following illustration shows a record containing the fields of information that can be found on a driver's license:

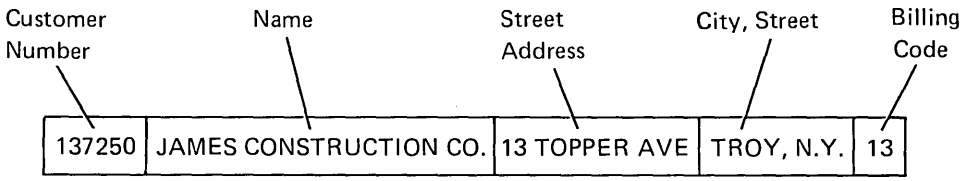


Each field is related to the record in that it contains information relating to the specific driver. A field is the amount of space set aside for each data item. The next illustration shows a record containing the fields of information found on a time card:



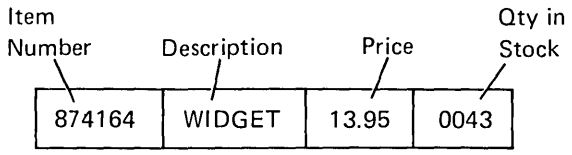
A group of records makes up a file. A 5110 data file contains records in a specific sequence just as a filing cabinet does.

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:

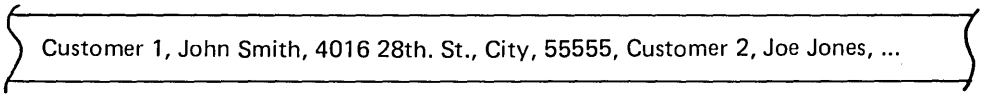


Organizing a File

An important part of any data processing job is file organization. File organization is the arrangement of records in the file. There are two types of files using the 5110: stream I/O and record I/O.

Stream I/O

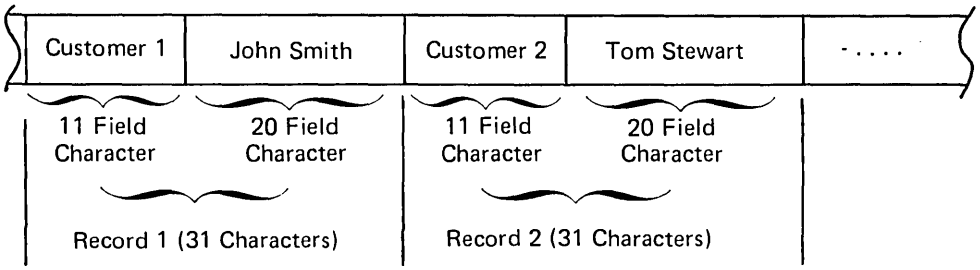
For stream I/O, all the data items are organized sequentially on the tape or diskette, with a comma used as the delimiter between fields. For example, a customer master file might be formatted as follows:



The fields are variable in length because only the exact number of characters is maintained. In order to read customer number 2 and the related fields, the 5110 must start at the beginning of the file and sequentially read each field until the desired customer information is read.

Record I/O

The corresponding fields of each record in a record I/O file must have the same length; no delimiters (commas) are required between fields. For example, a customer master file might be formatted as follows:



The record and field sizes are established as the application is designed (see Designing A Record in Chapter 10).

Unlike stream I/O files, record I/O files can be accessed in three ways:

1. *Sequential*. Each record is accessed in the same order they were written to the file.

2. *Direct*. Individual records can be accessed by specifying the record number (relative record number).

3. *Indexed*. An index is used to find an individual record in a file. Therefore, you do not have to know the relative record number of a record before you can access the record.

The next chapter describes the characteristics of the 5110 data file processing methods.

Chapter 9. Characteristics of Accessing Data Files

This chapter describes the characteristics of data files when the files are accessed:

- In sequential order
- In direct order by relative record number
- In direct order using an index

This chapter also discusses maintaining files.

SEQUENTIAL ACCESS

For both stream I/O and record I/O, a file can be accessed sequentially. That is, the records are accessed one after another in the order they occur. An example of a sequentially accessed file might be an employee master file. This file contains information needed for various reports concerning each employee, such as payroll checks. Because checks are processed by employee number, records are accessed in order. The lowest employee number is accessed and processed first and so on until the last record, the highest employee number, is accessed and processed.

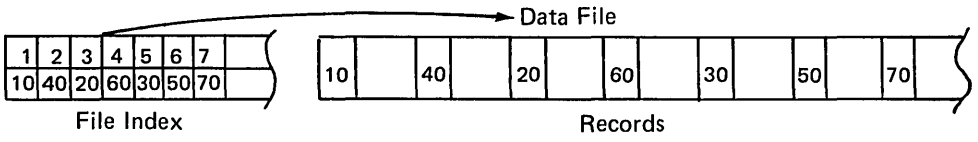
DIRECT ACCESS BY RELATIVE RECORD NUMBER

For record I/O, files can be accessed directly using the relative record number. This allows you to process records in the file faster than if you used sequential accessing. For example, assume you have an item master file that contains stock status information on 1000 items by item number. If you want to know the stock status of item number 500 in the file, direct accessing allows you to specify the record number containing the information. This record is then accessed directly and the information is available. However, if the file is sequentially accessed, you must read all of the preceding records before you can read the record that contains the information you need.

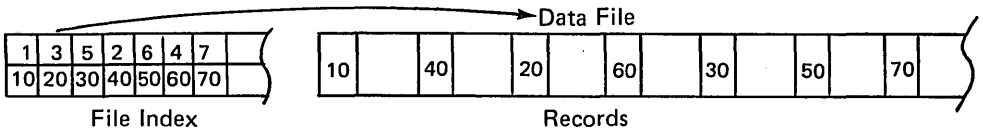
DIRECT PROCESSING BY INDEX KEY

For record I/O, files can be accessed directly using an index to locate the records in the file. The file index is created as the records are written to the file. The index contains a key, such as customer number or item number, and the relative record numbers of the record. When you want to process a specific record, you must specify the index key and the system accesses the proper record using the address of the record associated with that key.

After the index is created, the index can be sorted into ascending sequence. For example, as a file is created, the index and file are as follows:



However, after the index is sorted in ascending sequence, the index and file are as follows:



The diskette address of the record associated with each key remains the same. This allows you to access the data file in several ways.

Sequential Accessing by Key

When an indexed file is accessed sequentially by key, the keys are processed one after another in ascending order. Even if the records are not in order on the file, they are accessed in order using the index.

An indexed file can also be accessed sequentially, without using the index. Data records are accessed sequentially, that is, first record, second record, and so on, from the beginning of the file to the end of the file. However, if the records are not sorted first, they might not be in order.

Note: If you access an indexed file without using the index, and add or update records in the file, a key is not added to the key index for the added records, and existing keys within a record could be changed when you are updating a record.

Direct Accessing

Indexed files can also be accessed directly. This type of accessing also uses the index and is called direct accessing by key. Direct accessing by key permits processing of one particular record without regard to its relation to other records. You must first specify the key of the record. The key is then found in the index; the relative record number (adjacent to key) is used to locate the record; and the record is transferred to storage for processing. For example, records in a customer master file are to be updated to reflect current information. The transaction record number entries are not in order. The system finds the record by matching the customer number in the entry with the key (customer number in the index). The address, adjacent to this key, is then used to find the record.

Often an indexed file is used in several different jobs each of which requires a different accessing method. For example, during statement writing, a customer file may be accessed sequentially to allow cyclic statement writing. During a billing job, the same file may be accessed directly by key to allow the updating of specific master records. Then, during an aged trial balance job (each customer's outstanding balance is printed), the file may be accessed sequentially by key.

Indexed files can also be accessed by relative record number. This method of accessing requires that the file index be bypassed. Records in the data file can then be accessed by using a number (relative record number) indicating their relative position in the file.

MAINTAINING DATA 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 to a file after the file has been created. When records are added to a file, they are written at the end of the file. 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 may be necessary to keep the file in a particular order. In order to put the new records in the proper sequence, you must sort the file to create a new file containing the added records in the correct location.

When a record is added to an indexed file, the system checks to ensure that the record key is not a duplicate of a record key already in the file; if the key is not a duplicate, it is added to the end of the file. The keys of the added records and the keys of original records should be sorted, so that the keys of all the records in the file are in ascending sequence in the index. When the keys are in ascending order, the 5110 uses less time to search the index.

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 programs can check for the delete code; if the delete 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 file space. You can remove the deleted records by using a program to copy the records to be retained onto another file.

For an indexed file, you can also use the DELETE FILE statement to tag the record key in the index for the record to be bypassed. This does not alter the record in the master file.

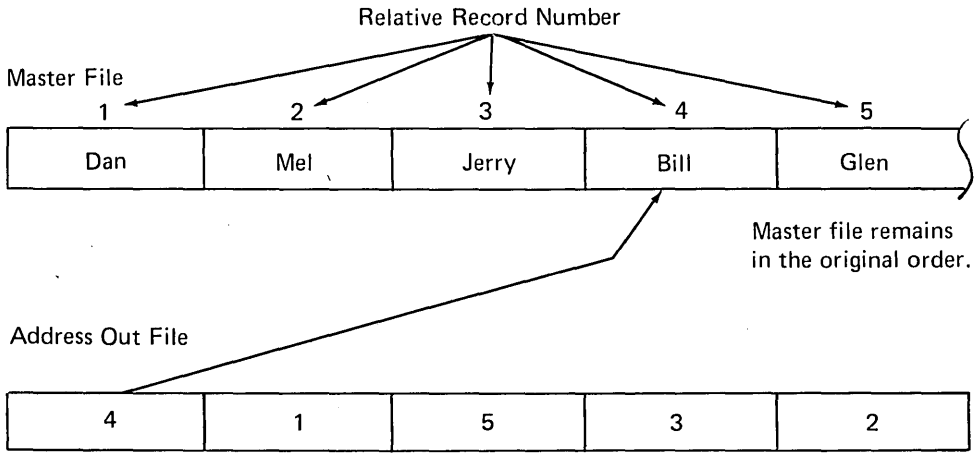
Updating Records

When you update records in a file, you can edit 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 (by the READ FILE statement) into storage, changed, and written back in its original location by the REWRITE FILE statement.

Reorganizing a File

After file maintenance activities are performed, it might be necessary to reorganize your file to increase processing efficiency and free file space. This is done by using a BASIC program to physically merge the added records in sequence with the records originally created, and to remove the records tagged for deletion by copying the existing file and writing it into a new file. During the copy, deleted records can be removed from the file, and records previously added to the file are copied into the new file in sequence with the original records. The old file can then be used to contain new data.

A Diskette Sort feature is provided to allow you to change the order of record I/O files. For more information about this program, see the *IBM Customer Support Functions Reference Manual*, SA21-9311. This feature allows you to sort the records and write them to a new file. Also, you can sort the file and write just the record numbers, which indicate where the records are located on the diskette, to a new file. This file, called an address out file, can then be used to access the original records. For example:



The following statements might be used in a program to access a file in order using the record numbers in the address out file:

```
 *
 *
 *
0030 OPEN FILE FL1,'D80',1,'NAMES',ALL
0040 OPEN FILE FL2,'D80',2,'ADDSORT',IN
0050 READ FILE USING 060, FL2,R
0060 FORM B4
0070 READ FILE FL1, REC=R,A$
 *
 *
 *
```

Chapter 10. Designing a Record and Determining File Size for Record I/O Files

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 determine field length, location, and name.

To illustrate these design considerations, a name and address file is used in this chapter. Each record in the file contains the following data:

Field	Size (number of positions)
Customer number	6
Name	20
Street address	20
City and state	20
Record code	2
Delete code	1
(Other fields)	47 (total)
	116 TOTAL

Determining Field Size

Field size depends on the nature of the data in the field. First, the length of the data may vary. In this example, name is 20 positions. The length of each customer's name varies, but 20 positions should be sufficient for the names. Secondly, all data in a field may be the same length. For example, customer number is six positions, and all six 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 name is planned as 15 positions, and a new customer has 19 characters in his name, a problem arises when his record is added to the file. To avoid this problem, try to estimate the largest length of the data that will be contained in a field. Use this length to determine field size.

Providing for a Delete Code

Records are not automatically deleted. You must place a delete code on a record with a BASIC REWRITE FILE statement. Then when the file is processed, your program can bypass the record.

For example, you might use the delete code to indicate that a customer is inactive and that his account information should not be processed when generating a report.

Record Expansion

It is often wise to allow for data to be added to a record. For example, suppose this name and address file were 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 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% is suggested.

Designing a Sample Record

Assume you are teaching a class and you decide to set up a record for each person. Each record will contain the person's name, his home address, test marks for five tests you plan to give, his average mark, and a code to indicate whether he is an honor student. The entries in the record might look like this:

ENTRY	CHARACTERS
NAME	25
ADDRESS	65
GRADES	15
AVERAGE	5
HONORS	1
	111 Total

Altogether, these entries take up 111 characters. You decide to include additional space in each record for possible entries to be added later, such as awards, special achievements, and remarks. Altogether, you decide to have a record with 128 characters in a file named CLASS with 128 positions for the record.

DETERMINING THE SIZE OF A FILE

To determine the size of a file, you must plan how many records will be in the file at a specified time.

To determine the number of records in a file, consider several factors. First, you must know how many records will be in the file when it is created. If the file already exists, perhaps as a card file, use the number of records in this file as a base.

You must also know whether records will be added or deleted. If additions are expected, how many records are expected, and how often will they occur? If records will be tagged for deletion, consider periodically removing them from the file. By removing records that you no longer need, you free diskette space and allow more records to be added.

Only after considering these factors and the applications that use the file can you determine the number of records in the file. For example, the customer name and address file will contain 6000 records at creation time. It is estimated that each month 200 records will be added and 80 records will be deleted. It is also planned that the deletion records will be removed once a month. At the end of six months the file will contain 6720 records (1200 records are added; 480 records are deleted).

6000	Records at creation
+1200	Records added in six months
7200	
- 480	Records deleted in six months
6720	Records in file after six months

This example points out another factor to consider. When determining the number of records in a file, consider expansion for a reasonable time into the future (at least six months). Of course, most files have deletions, and thus growth is usually slow. In a file where the number of additions and deletions are about the same, records tagged for deletion need be removed only when the disk space allowed for the file is filled.

Calculating File Space

To determine the file space, you must know the number of characters in the file. To calculate the number of characters in a file, multiply the number of records (allowing for expansion) by the length of each record. For the customer name and address file, there will be 6720 records in the file at the end of six months. Each record contains 128 characters. Thus, the number of characters in the file is calculated as:

6720	Number of Records in the File
x128	Number of Characters in Each Record
53760	
13440	
6720	
860,160	Total Characters in the File

and the file should be marked for 860K.

Calculating Index File Space

If the file is indexed, the system stores the index on a file. To determine the space needed for the index, you must know the size of the index entry (an index entry consists of a key and a diskette address). Key lengths vary, depending upon the application up to a maximum of 28 characters, but diskette addresses are always 4 characters long. Thus, the key entry is calculated as follows:

```
Key Entry = Key Field Length + 4
```

Note: The records in an index file must be 8, 16, or 32 characters long. Therefore, if the key entry is greater than 8 but less than 16, the index file record length is 16. Similarly, if the entry length is greater than 16 but less than 32, the index record length is 32.

In the name and address file described earlier in this chapter, the key field is customer number (#), and it is 6 characters long. In this case, the key entry is 10 (6 + 4 = 10) and the index file record length is 16 characters.

Now that we know the record size, we can calculate the storage required:

16*6,720 = 107,520

Thus, the index file should be marked for 105K (K = 1,024 bytes).

After you determine the amount of space the file requires, you can decide where to locate the file on the diskette. A diskette can contain several files, depending upon their size: therefore, you should document the files that are on each diskette using the UTIL PRINT command.

As you create more files, you can refer to the directory of a particular diskette to determine the amount of available space on that diskette.

Review–Calculating File Space

Calculation 1: Record Space

To calculate the space required for the records of a file, the following steps are necessary:

1. Multiply the number of records by record length to get the total number of characters.

2. Mark the file to the nearest number of K bytes.

Calculation 2: Index Space

To calculate the amount of space required for an index, the steps are:

1. Add 4 to the key field to get the length of the key entries.

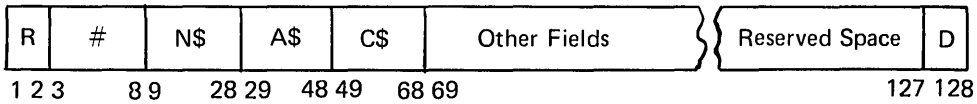
2. Determine index file record length, which must be 8, 16, or 32. Always assume the closest number that is not smaller than the length of the key entry to get index file record length.

3. Multiply the index file record length by the record count to get the character count.

4. Mark the file to the nearest number of K-bytes (characters).

DOCUMENTING RECORD LAYOUT

Documenting the record layout makes your BASIC program easier to write. The following example shows the layout of a customer master record:



where:

R = Record code

= Customer number

N$ = Customer name

A$ = Customer street address

C$ = City and state

D = Delete code

A record layout includes the order of the fields in the record, the length of each field, and the name of each field.

Record Length

A record may contain all predefined fields, or space may be reserved for data to be added to the record. In either case, all records in a particular file must be the same length. In your BASIC programs you must specify record length. Record length is the sum of the field lengths (including reserved space).

In the previous example, the sum of the fields is 116 positions. However, the record length is established at 128, thus 12 positions are reserved for data that might be added at a later time.

Chapter 11. Processing a Data File

This chapter discusses how to process stream I/O and record I/O data files.

PROCESSING STREAM I/O FILES

To process a stream I/O data file, you follow this sequence:

1. Open the data file.
2. Sequentially read from or write to the file.
3. Close the data file.

Opening and Closing Stream I/O Files

Files must be activated or *opened* before they can be used. A file can only be activated with an OPEN statement in the program. For example:

```
0010 OPEN FL1,'D80',2,IN
```

The word IN indicates that the file is to be used for retrieving data items from the file for use in the program. If the file were to be used for storing data, it could be opened explicitly as an output file with this statement:

```
0020 OPEN FL1,'D80',2,'NAME',OUT
```

Now, look at the following OPEN statement:

```
0210 OPEN FL3,'D80',5,'ITEM.MASTER',IN
```

For input/output operations, a file must be identified with a file reference of FL0-FL9. In the previous example, FL3. This file reference is used to identify the file when you are using GET or PUT statements (for example, GET FL3, A, B, C). After the file reference, the device is specified (for example, D80). A file number and/or file name can also be entered for an OPEN statement. For the previous OPEN statement, the file number is 5 and the file name is ITEM.MASTER. For diskette data files, a file name must be specified when the file is created. However, for tape files or reading any file, the file name does not need to be specified.

Normally, a file is deactivated or closed by the system after execution of your program. However, if you want to switch an input file to output (or vice versa) and continue to use it in the same program, you must explicitly deactivate it by using the CLOSE statement before reopening it. (If you did not use the CLOSE statement and attempted to use an output file for input or vice versa, execution of your program would be terminated.) CLOSE deactivates the file; a subsequent OPEN statement opens (reactivates) the file for its new use and repositions it at its beginning. For example:

```
0010 OPEN FL8,'D80',4,'ACCTS',OUT
0020 PUT FL8,D$,R$,A,B,C,D
0030 CLOSE FL8
0040 OPEN FL8,'D80',4,'ACCTS',IN
0050 GET FL8,D$,R$,A,B,C,D
```

In this example, the values assigned to the variables D$, R$, A, B, C, and D (statement 20) will be stored in file 4 (named ACCTS) on diskette drive 1. The file is then closed and reopened for input. Statement 50 then retrieves the variable values from the file. File reference code FL8 is used only to refer to the file opened in statements 10 and 40.

Writing to and Reading from Stream I/O Files

Stream I/O files can only be accessed sequentially. That is, you can only write or read records in sequential order starting from the beginning of the file. To do this, you use the PUT and GET statements. For example:

```
0080 OPEN FL1,'D80',2,'INTEREST',OUT
0090 PRINT 'ENTER PRINCIPAL'
0100 INPUT P
0110 FOR T=1 TO 10
0120 FOR R=1 TO 20
0130 A=P*(1+R/100)↑T
0140 PUT FL1,T,R,A
0150 NEXT R
0160 NEXT T
0170 STOP
```

Execute statements 130 and 140 two hundred times.

Write the values of T, R, and A to the data file.

The PUT statement instructs the computer to put the values contained in the variables T, R, and A into the file referenced by FL1.

Now, to read and print the data written to the file, you could use the following program:

```
0010 OPEN FL8,'D80',2,IN
0020 PRINT 'TIME','RATE','AMOUNT'
0030 FOR T=1 TO 200
0040 GET FL8,A,B,C
0050 PRINT A,B,C
0060 NEXT T
0070 STOP
```

Display the data under the appropriate title.

It is not necessary to use the same variable names as when the file was created. The important requirement is that the values in the file and the variables to which they are assigned must be the same type: arithmetic variables for arithmetic values, character variables for character values.

After the first GET is executed, the file is positioned at the next value. In the previous example, the GET statement is executed 200 times to access all the data previously stored.

Notice what happens when an input file is closed and reactivated as an output file.

```
0020 OPEN FL4,'D80',2,'AF',IN
0030 GET FL4,A,B,C,D,E
0040 B=A
0050 A=36
0060 C=C+B
0070 CLOSE FL4
0080 OPEN FL4,'D80',2,'AF',OUT
0090 PUT FL4,A,B,C
```

A previously created file named AF is activated for input. In statement 30, five values are made available to the program from file 2. In statements 40 through 60, new values are acquired for A, B, and C. Statement 70 deactivates AF, and statement 80 re-opens the file for output. Statement 90 places the new values for A through C into the file. All of the old values in the file are lost.

Repositioning Files

Occasionally you may have to use an input file or an output file more than one time in the same program. The RESET statement allows you to reposition the file without deactivating it (deactivation is necessary only when the function of a file is changed from input to output or vice versa). For example:

```
0050 GET FL9,X,Y,Z,Q,R,S
  .
  .
  .
0100 RESET FL9 <------------ Repositions file to
0110 GET FL9,X,Y,Z,Q,R,S     its beginning
  .
  .
  .
0150 RESET FL9
0160 GET FL9,X,Y,Z,Q,R,S
```

Between statements 50 and 100, the variables X, Y, Z, Q, R, and S could be used in one set of calculations and their values changed. Repositioning the file to the beginning permits the original values in the file to be made available and put into variables X, Y, Z, Q, R, and S again for different calculations or uses between statements 110 and 150, and again between 160 and the end of the program.

To add data to the end of the file, you can reset the file to its end by using the RESET statement with the END keyword:

```
0200 RESET FL1 END
```

This statement positions FL1 to the end of the last data item in the file. PUT statements appearing after statement 0200 place additional data into the file. In effect, RESET FLX END changes an input file to an output file. In this case, the file must be open for input before you use the RESET END statement.

Input/Output Error Handling

Certain error conditions can occur while you are processing files. As an example, when reading through a file, you need to take action after the last item is read; otherwise the computer will terminate the program. The EOF (end of file) clause can be written in the GET statement to branch to another program statement when the end of the file is reached.

A GET statement with an EOF clause could look like this:

```
0040 GET FL6,X,Y,Z,EOF 100
```

This statement directs the computer to statement 100 when the end of the file is reached. At statement 100, you could end the program, or close the file and continue processing, or perform any number of actions. The important thing is that specifying the EOF clause allows you to retain control of program execution.

The EOF clause can be specified on the PUT statement as well. Note that if an EOF condition occurs, not all of the output data may have been written into the file.

These are other error handling clauses:

Clause	Meaning
IOERR n	Branch to the statement numbered *n* if a hardware malfunction prevents reading or writing of a record. IOERR can be specified on the GET and PUT statements.
CONV n	Branch to the statement numbered *n* if a conversion error occurs while a data item is being assigned, for example, if an attempt is made to read character data into a numeric variable. CONV can be specified on the GET statement but not on the PUT statement.

Instead of writing these error handling clauses on many GET and PUT statements throughout your program, you can write them on one or more EXIT statements. An EXIT statement is used in conjunction with many input/output statements to group error handling in one place. The statement could look like this:

```
0080 EXIT EOF 100,IOERR 150,CONV 200
```

This statement tells the computer to branch to statement 100 when the end of the file is reached, to branch to statement 150 if a hardware error is encountered, and to branch to statement 200 if a data conversion error is encountered.

ACCESSING RECORD I/O FILES

You can access record I/O files by three methods: sequential, direct, and indexed.

The sequential access method is one in which the records are accessed in the order in which they are entered. To use an example of the 50 states, if you enter the records in alphabetic order, the first record is Alabama, then Alaska, Arizona, Arkansas, and so on. If you enter them in geographic order, say with the New England states first, the order is Maine, New Hampshire, Vermont, and so on. In either case, all records are retrieved sequentially in the same order that they were entered.

In a record-oriented file, each record has a record number relative to the first record. If the 50 states are stored alphabetically, the Arkansas record has a relative record number of 4. The direct access method can be used to retrieve records directly by record number.

An indexed access method is one in which each record is stored with a unique identification called a key. If the 50 states were stored with a key (for example, the key could be the name of the state), you can tell the 5110 which key to look for. The computer looks through an index until it finds the particular key and then retrieves the corresponding record from the master file. Thus with an index, each record can be retrieved directly.

To process a record I/O file, you follow this sequence:

1. Open the data file.

2. Access the file sequentially, directly, or indexed directly.

3. Close the data file.

Opening and Closing Record I/O Files

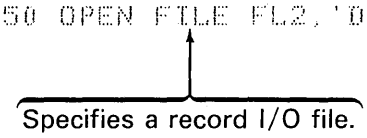
Record-oriented files, like stream-oriented files, must be opened explicitly. A record-oriented file is opened explicitly through the OPEN FILE statement. As you may recall, for stream-oriented files, OPEN is specified with the keywords IN for input or OUT for output. Record-oriented files are opened in the same way for input and output. If ALL is specified, the file can be accessed for both input and output without closing and reopening.

In addition, the RECL= clause (record length) must be specified after OUT when creating record-oriented files to specify the length (number of characters) of the record being written.

CLOSE FILE is used to close files the same way for record-oriented files as CLOSE is used for stream-oriented files. If the statement is not present, the system closes the file at the end of program execution.

Following is an example of an OPEN FILE statement for a record I/O file:

```
0150 OPEN FILE FL2,'080',4,'NEW.ACCOUNTS',OUT,RECL=120
```



Notice the period (.) in the file name NEW.ACCOUNTS. No blanks are permitted in the file name.

If you are going to use the indexed access method, you must also open a file for the index. For example:

```
0210 OPEN FILE FL3,'080',6,'CUSTOMERS',OUT,RECL=128
0220 OPEN FILE FL3,'080',2,'INDEX',OUT,KEY,KP=1,KL=25
```

The file reference must be the same for the master and the index file; however, the files must be in different locations on the media (in this example, files 6 and 2). After statement 220 is executed, the 5110 automatically creates an entry in the index file when a record is written to the master file. The KP= and KL= parameters describe the starting location and number of characters of the record in the master file to be used as a key in the index.

Note: If the parameter SEQ is specified in the open statement for a data file, that file can be used as a data exchange file with other systems. However, this file should only be accessed sequentially. Direct access of the file might not access the desired record.

Writing to and Reading from Record I/O Files

Creating a Record I/O File

The WRITE FILE statement is the record-oriented counterpart to the PUT statement. For example, at the beginning of the school session, the only information available to you for each student is his name and address. You could write one WRITE FILE statement to enter the name and address for each student like this:

```
0050 WRITEFILE FL1,'BUTLER, J.S.','323 W. 76 STREET, NEW YORK'
```

You could also write one generalized WRITE FILE statement using two character variables for the name and address, like this:

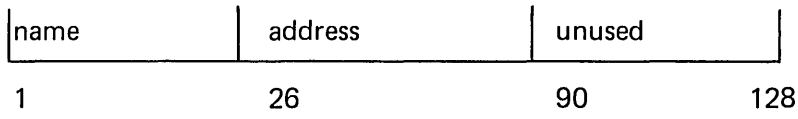
```
0050 WRITEFILE FL1,N$,A$
```

This statement would enter the values of the two variables N$ and A$.

This DIM statement should be included in the program to assign a length of 25 to N$ and 65 to A$:

```
0010 DIM N$25,A$65
```

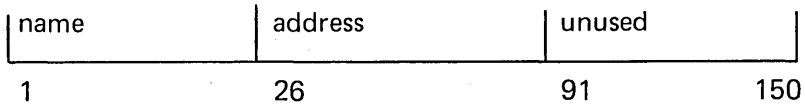
Each record written by the WRITE FILE statement would be arranged in the file this way:



Note that this WRITE FILE statement writes 90 positions of the record. Unassigned record space is filled with blanks. Thus, record positions 91 through 128 are blank. The WRITE FILE statement contains a USING clause with the statement number of the FORM statement, and the FORM statement describes how the entries are to be formatted into the record. The combination of WRITE FILE and FORM statements could look like this:

```
0050 WRITEFILE USING 55,FL1,N$,A$
0055 FORM POS1,C,POS26,C
```

This FORM statement says that, beginning at position 1 in the record, the character variable N$ is to be written; beginning at position 26 of the record, the character variable A$ is to be written.



The following program shows how you could enter the names and addresses of the students into the file named CLASS.

```
0010 OPEN FILE FL1,'D80',1,'CLASS',OUT,RECL=128,SEQ
0020 DIM N$25,A$65
0030 PRINT 'ENTER NAME'
0040 INPUT N$
0050 IF N$='LAST' GOTO 110
0060 PRINT 'ENTER ADDRESS'
0070 INPUT A$
0080 WRITEFILE USING 90,FL1,N$,A$
0090 FORM POS1,C,POS26,C
0100 GOTO 30
0110 CLOSE FILE FL1
```

The program is constructed to recognize the word LAST as the end of input; therefore, the last input item should be coded 'LAST'. Your input could look like this:

```
'BUTLER,J.S.'
'323 W. 76 STREET, N.Y., 10023'
'COOK, A.B.'
'3062 STREET, WEST NEW YORK, N.J., 07094'
  .
  .
  .
'SMITH, C.A.'
'228 E. 55 STREET, N.Y.,10022'
'YOUNG, W.'
'3230 145 STREET, FLUSHING, N.Y., 11358'
'LAST'
```

After the records are entered, the first record in CLASS would look like this:

```
BUTLER, J.S.                323W. 76 STREET, N.Y., 10023
```

After the file has been created, if you decide to add more records, say for a new student who registers late, the WRITE FILE statement can be used to enter additional records. No RESET statement is necessary as with stream-oriented files; the WRITE FILE statement automatically positions a file at its end. Note that additional records would not be sorted but would be entered in place at the end of the file.

Now, assume an index file was also specified for the previous example:

```
0015 OPEN FILE FL1,'D80',2,'INDEX',OUT,KEY,KP=1,KL=25
```

Use the characters in positions 1 through 25 as the key.

When formatting the key field, you should exercise care in putting the key into the proper position in the file. For purposes of simplicity, these examples use the first 25 record positions for the key. The occasion may arise, however, when you might have a file with the key starting in a position other than 1. By careful use of POS , you can assure that the key will be properly located. Also, you can use the intrinsic function KPS (FLX), to find the position, relative to 1, of the start of an embedded key in the file referenced by FLX, and you can use the intrinsic function KLN (FLX) to find the length of the key. These functions are described in the *5110 BASIC Reference Manual*.

After the records are entered into CLASS, additional records can be added and will be stored in key-indexed order.

Reading Records from a Record I/O File

A record I/O file can be read sequentially, directly using a relative record number, directly by key index, or sequentially by key index. The method you use depends upon the requirements of your application. Following is a description of the four ways to read a record I/O file.

Sequentially

The READ FILE statement is used to sequentially read a record I/O data file. For example:

```
0010 OPEN FILE FL1,'D80','PAYROLL',IN
   *
   *
   *
0090 READ FILE USING 100,FL1,A$,B$,C,D$
0100 FORM C10,C20,NC10,C5
   *
   *
   *
   *
   *
   *
   *
0180 GOTO 90
```

Notice that the file is first opened as an input file with FL1 as the file reference. The READ FILE statement uses the same file reference. The FORM statement (statement 100) specifies the format of the record.

Each time the READ FILE statement is executed, the system reads the data in the next sequential record in the file. Thus, the records are read in the same sequence they were written to the file.

It is not necessary to read the items in the same order in which they appear in the record. For example, the statement could be written:

```
0090 READFILE USING 100,FL1,C,D$,B$,A$
0100 FORM POS31,NC10,POS41,C5,POS11,C20,POS1,C10
```

Nor is it necessary to read all the items in a record. If you were interested only in obtaining name information, you could use this READ FILE and FORM combination:

```
0090 READFILE USING 100,FL1,A$
0100 FORM POS1,C10
```

This combination might be helpful when you wish to insert test marks for each student. You could read through the file sequentially, obtain each student's record, display his name on the screen for verification, and enter the corresponding mark.

The READ FILE statement, like the GET statement, can contain an EOF clause to transfer control when the end of the file is reached. In the program shown below, the READ FILE statement causes program control to branch to statement 100 at the end of the file, which is used to print a message.

This program shows how you can read each student's record to insert a test mark. The program also introduces the REWRITE FILE statement (see Updating Records), which is used to update an existing record, and shows how OPEN and CLOSE statements can be used with record-oriented files.

```
0020 DIM J$25
0030 OPEN FILE FL1,'D80',3,'CLASS',ALL
0040 READFILE USING 45,FL1,J$,EOF 100
0045 FORM POS1,C
0050 PRINT J$
0060 INPUT G
0070 REWRITEFILE USING 75,FL1,G
0075 FORM POS101,PIC(ZZ#)
0080 GOTO 40
0100 PRINT 'END OF FILE-LAST RECORD READ'
0120 STOP
```

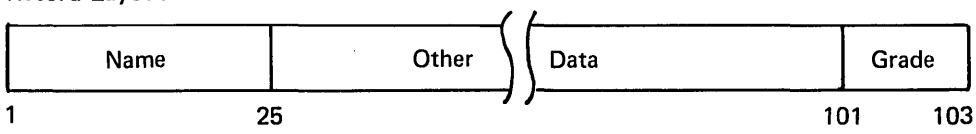
Opens the file for input and output.

Branch to statement 100 when all the records are read.

Read name from the file.

Update the record with the grade entered from the keyboard.

Record Layout



Statement 30 opens the file. ALL is a special keyword used with record-oriented files to indicate that *both* input and output operations can take place on the file. ALL is required if any rewriting operations are to take place.

Statements 40 and 45 obtain the name information from the file. Statement 50 displays the name, allowing you to verify it and enter the corresponding test mark in statement 60. Statement 70 is the REWRITE FILE statement, which enters one data item into the record just read, the numeric variable *G*. Statement 75 says that the variable is to be entered beginning at position 101 of the record, in the format PIC(ZZ#), three digits with leading zeros suppressed. The remaining statements cause the program to cycle through all the records and close the file after the last record is handled.

Directly Using Relative Record Number

To retrieve records directly by relative record number, specify the REC= clause in the READ FILE statement. If, for example, you want to access the Nth record, specify:

```
70READ FILE USING 75, FL1, REC=N, C$, A$
```

The following example shows this method of record retrieval:

```
0010 DIM D$20
0020 OPEN FILE FL1,'D80',2,'MASTER',IN
0040 PRINT 'INPUT REC'
0050 INPUT K
0051 IF K=0 GOTO 90
0060 READFILE USING 70,FL1,REC=K,X,D$
0070 FORM NC10,C20
0080 PRINT X,D$
0090 STOP
```

In this example, file number 2 (referenced as FL1) on diskette drive one is opened for input. Statement 40 requests keyboard input, which is assigned to variable K in statement 50. Statement 51 tests for the end of the program. In statement 60 variables X and D$ are read from the file referenced as FL1, using the value of the variable K to access the record in the file MASTER. Statement 70 specifies the format for the data being read. The data is displayed (statement 80), after which the program branches back to request keyboard input again.

Directly Using an Index

If an index file was created, records can be read from the master file using an index. In this case, both the master file and the index file must be opened:

```
0010 OPEN FILE FL2,'D80',2,'GRADE',IN
0020 OPEN FILE FL2,'D80',3,'INDEX',IN,KEY
```

Specifies direct access by key index.

If you wanted Smith's record, you would specify his name in the KEY clause in the READ FILE statement:

Direct access using a key.

```
0065 N$='SMITH, C.A.'
0070 READ FILE USING 75,FL2,KEY=N$,F$,G$
0075 FORM * * *
```

The 5110 will search for the record whose key matches in the index file, then will read the values from the master file record into the variables F$, and G$.

The following example shows this method of record retrieval.

```
0010 DIM D$20,K$10
0020 OPEN FILE FL1,'SYS',2,'MASTER',IN
0030 OPEN FILE FL1,'SYS',1,'INDEX',IN,KEY
0040 PRINT 'INPUT KEY'
0050 INPUT K$
0060 READFILE USING 70,FL1,KEY=K$,X,D$
0070 FORM NC10,C20
0080 PRINT X,D$
0085 GOTO 40
0090 STOP
```

In this example, the specific record in file number 2 (referenced as FL1) is selected by the key value you enter in statement 50. If, however, the computer cannot find the key, it indicates an error unless you instruct it to take alternative action. For example, if you enter the key incorrectly (say you spelled the name SMIHT), the match would not be found. To protect program execution, include the NOKEY clause on the READ FILE statement:

```
0065 N$='SMITH, C.A.'
0070 READ FILE USING 75,FL2,KEY=N$,F$,G$,NOKEY 200
0075 FORM * * *
```

The NOKEY clause tells the computer that if the matching key cannot be found, the program should branch to statement number 200. The NOKEY clause for indexed files is similar to the EOF clause for sequentially accessed files; it permits you to retain control of program execution if a particular condition arises.

Records can be read sequentially by key using the index by opening the index file and master file and specifying the READ FILE statement without the KEY parameter.

Updating Records in a Record I/O File

Part or all of a record in a record I/O file can be updated using the REWRITE FILE statement. When updating a record I/O file, the OPEN statement for that file must specify the ALL parameter. For example:

```
0020 OPEN FILE FL2, 'D80', 3, 'MASTER', ALL
```

The following sample program, which adds telephone numbers to records in an existing name and address file, shows how you can use the REWRITE FILE statement to update a record I/O file:

```
0010 REM
0020 REM
0030 DIM N$25,A$65,T$12
0040 OPEN FILE FL2,'D80',3,'MAIL.LIST',ALL
0050 READFILE USING 60,FL2,N$,A$,EOF 120
0060 FORM C25,C65
0070 PRINT N$
0080 INPUT T$
0090 REWRITEFILE USING 100,FL2,T$
0100 FORM POS91,C12
0110 GOTO 50
0120 PRINT 'END OF JOB'
0130 STOP
```

When the file was originally created, each record contained space available for the telephone number and other data. After a record is read (statement 50), the name is displayed (statement 70) and the telephone number can be entered (statement 80). Once the telephone number is entered, the record in the file is updated in positions 91 through 103 (statements 90 and 100) and the next sequential record is read. This process continues until the last record in the file is processed. After the last record is processed, an end of file (EOF) condition occurs and the program branches to statement 120.

If there is an index for the master file, you can access and update individual records without processing the file sequentially. For example, you could use the following program to update the telephone numbers of specific customers in an already existing name and address file:

```
0010 DIM N$25,A$65,T$12
0020 OPEN FILE FL2,'D80',3,'MAIL.LIST',ALL
0030 OPEN FILE FL2,'D80',4,'INDEX',ALL,KEY
0040 INPUT K$
0050 IF K$=' ' GOTO 120
0060 INPUT T$
0070 REWRITEFILE USING 80,FL2,KEY=K$,T$,NOKEY 100
0080 FORM POS91,C12
0090 GOTO 40
0100 PRINT 'NO MATCH FOUND'
0110 GOTO 40
0120 STOP
```

When this program is run, statements 40 and 60 request the customer's name and new telephone number. Then, if the name is found in the index file (KEY=K$), the master file is updated with the phone number (T$). If a name match is not found, the message NO MATCH FOUND is displayed (NOKEY=100), and the program requests the next name and telephone number. If a ' ' is entered as the name, the program branches to statement 120 and stops.

If the KEY clause (KEY=) is used in the REWRITE FILE statement, no READ FILE statement is required to retrieve the record first. If the KEY clause is specified, the record matching that key is brought in from the file; thus, the REWRITE FILE statement with a KEY clause retrieves as well as rewrites.

The REWRITE FILE statement can write over existing data or unused portions of a record, but must not change the contents of the field containing the key information. Fields not written over remain unchanged.

As another example of REWRITE FILE, assume that during the school term you give the students an extra credit project; their final grade will be raised by five to ten points depending on the quality of their work. Before the end of the term, you add in the extra credit for those students who handed in the project. The short program below illustrates how the REWRITE FILE statement can be useful in updating the records.

```
      .
      .
      .
0090 DIM N$25
0100 PRINT 'ENTER STUDENT''S NAME AND EXTRA CREDIT MARK'
0110 INPUT N$,E
0120 IF N$='LAST' GOTO 170
0130 REWRITEFILE USING 135,FL2,KEY=N$,E,NOKEY 150
0135 FORM POS140,PIC(Z#)
0140 GOTO 100
0150 PRINT 'NO MATCH FOUND FOR',N$
0160 GOTO 100
0170 STOP
```

Statement 100 prompts you for input information. Statement 110 accepts the student name in N$ and the mark in E. Statement 120 tests whether the end of input has been reached; assume the last input data item should have the word LAST as the student's name. Statement 130 enters the mark recorded in E into the file after the key has been matched with the name in N$. Statement 135 formats the mark into positions 140 and 141.

More Information About Processing Record I/O Files

Deleting Records

Records in an indexed file can be made unavailable with the DELETE FILE statement specifying the key of the record to be deleted. For example:

```
0090 DELETE FILE FL2,KEY=N$,NOKEY 130
```

This statement would delete (by modifying the index file) the record whose key matched the character value in N$, or would branch to statement 130 if the key could not be matched. The actual record is not removed. The record key in the index file is flagged, making the record inaccessible by key.

Repositioning Files

The RESET statement can reposition a file to its beginning. If RESET contains a KEY clause, the file is repositioned to the particular record associated with that key. If the RESET statement contains a REC= clause, the file is repositioned to the record specified by the REC= clause.

Error Clauses on the EXIT Statement

For record I/O files, the EXIT statement can specify these clauses in addition to the other clauses available:

- NOKEY, to transfer control if no key satisfying a KEY clause can be found.
- NOREC, to transfer control if the relative record number specified by the REC= clause cannot be found.
- DUPKEY, to transfer control if a key specified for a new record already exists in a file.

An EXIT statement specifying all error handling clauses could look like this:

```
0130 EXIT EOF 300,IOERR 320,CONV 350,NOKEY 130,DUPKEY 200
```

When using the EXIT statement, remember to include an EXIT clause on each appropriate input/output statement. For example, to refer to the EXIT statement above, the DELETE FILE statement previously illustrated could be written:

```
0090 DELETE FILE FL2,KEY=N$,EXIT 180
```

The FORM statement used with record-oriented files is similar to that used with PRINT USING in the following ways:

- Both contain the C character specification.
- Both contain the replication factor (see the *IBM 5110 BASIC Reference Manual.)*
- Both contain the PIC numeric specification, with the same digit specifiers and insertion characters.
- Both contain the format control specifications X and POS.
- Both contain character constants (see the *IBM 5110 BASIC Reference Manual.)*

The FORM statements used are different in the following ways:

- With record-oriented files, the FORM statement does not contain the SKIP format control, because there is no need for a skip operation.
- With record-oriented files, numeric data can be formatted using other specification codes besides PIC. Additional specification codes are:

  NC
  PD
  S
  L
  B

B, NC, and PD are used to store and retrieve numeric data in special internal formats. Except for one use of NC, they are not further discussed here; additional material on these codes can be found in the *BASIC Reference Manual* under *'FORM Statement.'*

The NC Specification

The one use of NC applicable to this discussion is in its relationship to PIC. PIC can be used only in output operations; thus, it can appear in FORM statements related to WRITE FILE and REWRITE FILE statements, but not in those related to READ FILE or REREAD FILE statements. To read data that was written using PIC, NC is used, specifying the number of positions in the record to be read. For example,

NC4

would read four positions of a number.

If a number were written using this PIC specification:

PIC(###) or PIC(ZZ#)

the NC specification to retrieve it would be:

NC3

To retrieve only the first two of these digits, you would specify NC2.

Earlier, this FORM statement was used to enter the two-digit numeric variable E into the file called GRADE:

```
0135 FORM POS140,PIC(ZZ#)
```

To retrieve that value, you could use this FORM statement:

```
0055 FORM POS140,NC2
```

NC can also specify the number of decimal digits in a number, in the following manner:

NC5.2

This specification says that five positions are to be read, and a decimal point is to appear before the two rightmost digits. That is, the five positions could look like this:

12.34	would be read as 12.34
1.234	would be read as 12.34
11234	would be read as 112.34

If an item were written using this PIC specification,

PIC(####.##)

The NC specification to retrieve it would be:

NC7.2 or NC7

The first number specified in NC is the field width, that is, the total number of characters to be read, including digits, decimal points, commas, dollar signs, etc. The second number is the number of decimal digits. The following are examples of how PIC and NC can be used in combination:

If PIC were specified:	NC would be specified:
PIC(###.##)	NC6.2 or NC6
PIC(ZZZ.##)	NC6.2 or NC6
PIC($$,$$$.##)	NC9.2 or NC9
PIC(ZZBZZBZZ)	NC8 or NC8.0

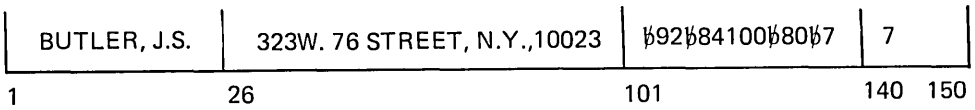
The S and L Specifications

The specification S indicates that an item in a record is in short-form precision. A number in short-form precision takes up four positions in a record. If S is specified for an input operation, the value in the record is moved to the variable specified in the READ FILE or REREAD FILE statement; if the program is in long form precision, such a value is extended to long-form. If S is specified for an output operation, a short-form value is written from the variable specified in the WRITE FILE or REWRITE FILE statement into the record.

The specification L indicates long-form precision and is the long-form counterpart to the S specification. A number in long-form precision takes up eight positions in a record.

For an input operation, the value in the record is moved to the variable specified in the READ FILE or REREAD FILE statement; if the program is in short-form precision, long-form items are truncated to short-form before being used. For an output operation, a long-form value is written into the record from the variable specified in the WRITE FILE or REWRITE FILE statement.

After all the marks for five tests and the extra credit for the project have been entered into the file GRADE, the first record in the file could look like this:



If you wanted to print the final mark and the honors status, you could use this program:

```
0010 DIM G(5),M$1,N$25
0020 PRINT USING 25,FLP,'FINAL MARK','HONORS'
0025 FORM POS6,C,POS35,C,POS50,C,SKIP2
0030 OPEN FILE FL2,'DSO',2,'GRADE',ALL
0050 READFILE USING 55,FL2,N$,MATG,E,EOF 110
0055 FORM POS1,C,POS101,5*NC3,POS140,NC2
0060 A=SUM(G)/5+E
0065 IF A≤100 GOTO 70
0067 A=100
0070 M$='+'
0071 IF A≥90 GOTO 80
0072 M$=' '
0080 PRINT USING 85,FLP,N$,A,M$
0085 FORM POS6,C,POS35,PIC(ZZ#.#),POS50,C,SKIP1
0090 REWRITEFILE USING 95,FL2,A,M$
0095 FORM POS130,PIC(ZZZ.Z),POS135,C
0100 GOTO 50
0110 STOP
```

Statement 10 defines an arithmetic array, G, with five members, a character variable, M$, with one character, and a character variable, N$, with 25 characters. The array G is to hold the five marks for each student, M$ is to hold the honors character, either a + or a blank, and N$ is the name field.

Statements 20 and 25 format a printed heading. Statement 30 opens the file for input, output, and updating operations.

Statement 50 reads the file according to the format shown in statement 55. Remember that although GRADE is a key-sequenced file, its records can be read in sequential order if the KEY clause is not specified. From statement 55 we can determine that the items being retrieved are the name, placed into N$, five sequences of three digits (the five marks beginning in position 101), placed into the array G, and a two-digit number for extra credit, placed into E.

Statememt 60 sums the five marks, divides the sum by 5 to find the average, then adds in the extra credit recorded in E, and puts the resulting value into A.

Statements 65 through 72 reduce any mark that exceeds 100 and analyze the value of A. If the value equals or exceeds 90, a plus sign, indicating honor student, is placed into M$. If the value of A is less than 90, M$ is assigned a blank.

Statements 80 and 85 print the student's name (N$), the final mark (A), and the honors code (M$).

Statements 90 and 95 enter the final mark and the honors code into the record, beginning in positions 130 and 135, respectively.

Statement 100 branches back to statement 50, and the next record is read. After all records are read, the program ends.

Output from this program could be the following:

Print Position	**6**	**35**	**50**
	NAME	FINAL MARK	HONORS
	BUTLER, J.S.	92.2	*
	COOK, A.B.	82.0	
	°		
	°		
	°		
	SMITH, C.A.	84.0	
	YOUNG, W.	97.0	*

Summarizing Record-Oriented Statements

The OPEN FILE statement explicitly opens a record-oriented file. If IN is specified, the file is opened for input; if OUT is specified, it is opened for output; if ALL is specified, it is opened for both operations. If the KEY clause is specified, an index file is associated with a master file.

The WRITE FILE statement writes a record. In a directly or sequentially accessed file, each record is stored in the order in which it is entered. In an indexed file, each record is stored in the order in which it is entered, and the record key is stored along with the relative record number in the index file. When you are retrieving or writing records by key, performance is improved if the index file is sorted into order by key.

The READ FILE statement reads a record. In a sequentially or directly accessed file, each record is read sequentially, or directly by relative record number when the REC= clause is specified. In an indexed file, each record is read sequentially if the associated index file is not open. If the index file is open and no KEY clause is specified, the records are read sequentially by key. If KEY is specified, the record having a matching key is read.

The REREAD FILE statement makes the last record previously read available again, regardless of whether the record was read sequentially or by key.

The REWRITE FILE statement alters an existing record, provided that the file was opened with the OPEN FILE statement specifying ALL. In a file accessed sequentially, the last record is read and rewritten. In an indexed file accessed sequentially by key, the last record read is read and rewritten if no KEY clause is specified in the statement. If KEY is specified, the record having a matching key is read and then rewritten. If the REC= clause is specified, the record with a matching number is read and rewritten.

The RESET FILE statement repositions a file to its beginning. In a file accessed sequentially, if a KEY or REC= clause is specified, the file will be repositioned to the particular record associated with that key, or record number.

The DELETE FILE statement deletes a record from an indexed file. The KEY clause is required in order to identify the record being deleted.

The EXIT statement specifies the statement number to which control should be given if a particular input/output error occurs. The error key keywords that can be written in the statement are EOF, IOERR, CONV, NOKEY, NOREC, and DUPKEY.

The CLOSE FILE statement explicitly closes a record-oriented file.

The FORM statement specifies the format of fields in record-oriented files.

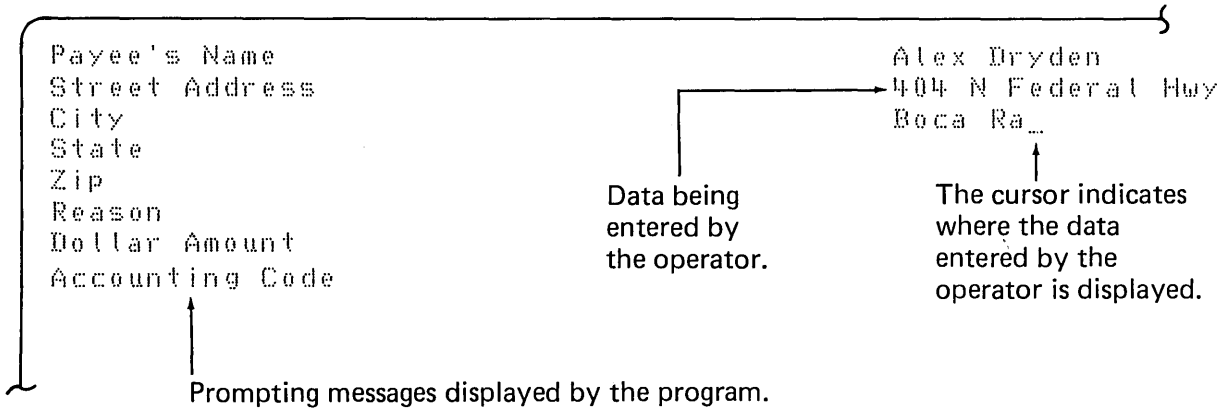
Chapter 12. Control of Your 5110

This chapter discusses the following topics concerning controlling your 5110:

- Using the display screen for input and output
- Using procedure files to replace keyboard input
- Using the system control functions (FILE FLS)
- Using the UTIL command

USING THE DISPLAY SCREEN FOR INPUT AND OUTPUT

You can use the WRITE FILE and READ FILE statements to write and read data from anywhere on the top 14 lines of the display screen. This allows you to use different screen formatting and data entry techniques. For example, the screen could be formatted as follows:



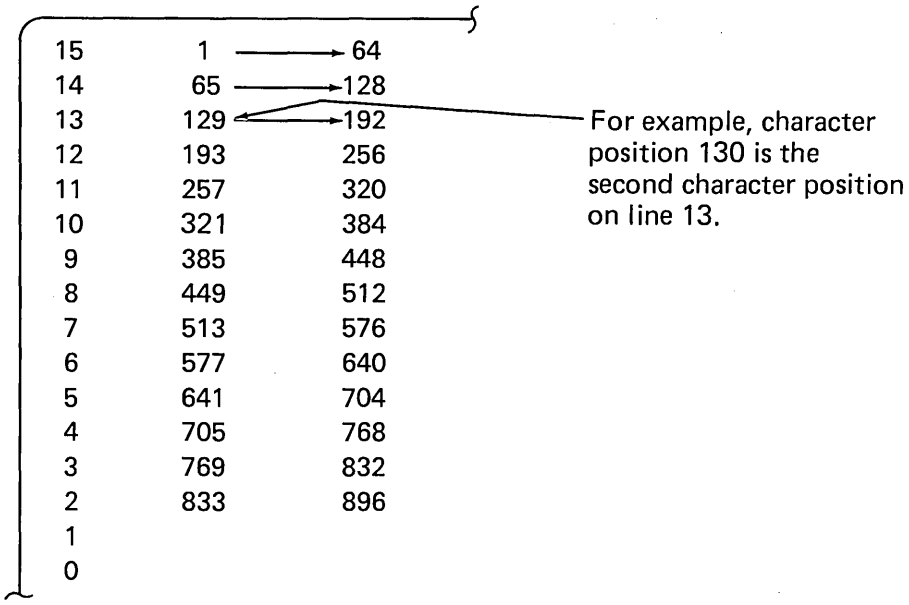
When used for input and output, the display screen is treated as a record I/O file, similar to a record I/O file on tape or diskette. You must open the display screen for input/output using device 002, for example:

```
0100 OPEN FILE FL1,'002',ALL
```

Specifies read and write operations

Input/output to the display screen

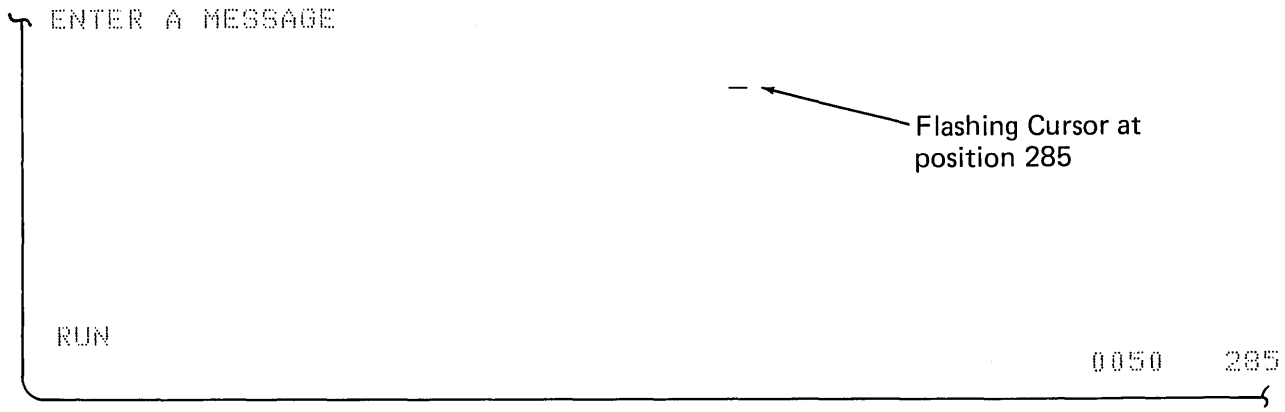
The character positions on the display screen are numbered as follows:



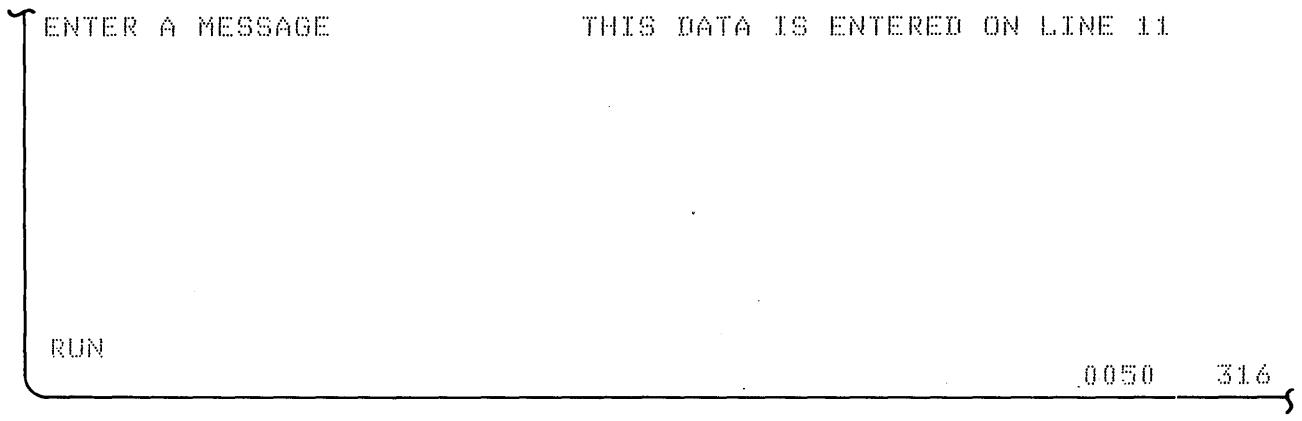
The following is a sample program that writes data to and reads data from the display screen:

```
0010 DIM A$50
0020 OPEN FILE FL1,'002',ALL
0030 WRITEFILE USING 40,FL1,'ENTER A MESSAGE'
0040 FORM POS257,C15,POS285
0050 READFILE USING 60,FL1,A$
0060 FORM POS285,C50
0070 REWRITEFILE USING 80,FL1,'THE MESSAGE IS:',A$
0080 FORM POS769,C15,X5,C50
0090 STOP
```

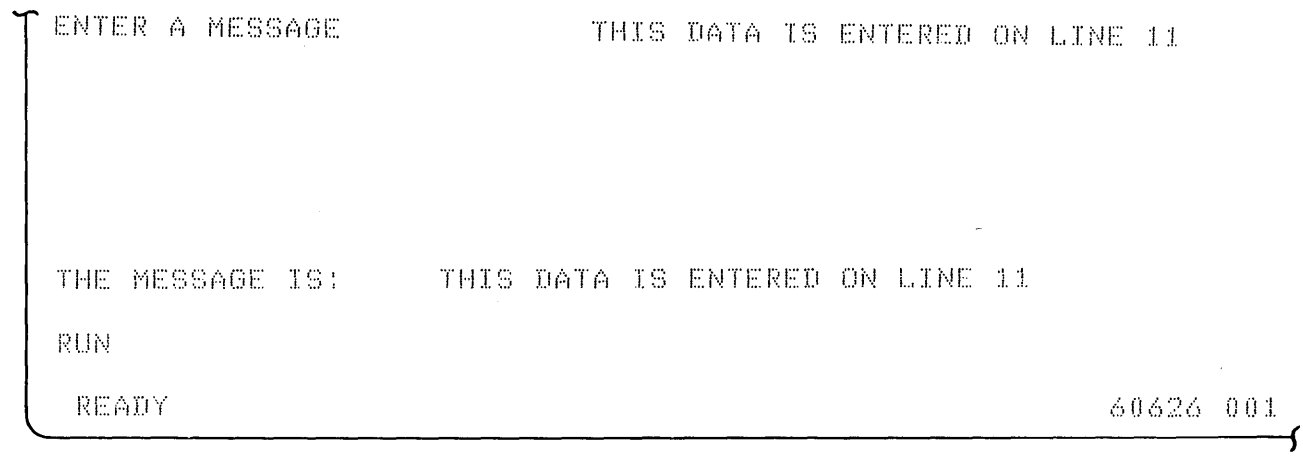
When this program is run, statements 030 and 040 write ENTER A MESSAGE starting at position 257 (line 11) on the display screen. The cursor is then placed at position 285 as specified in statement 0040 (POS 285), and the program waits for input from the keyboard. The display screen looks like this:



Now, when a message is entered from the keyboard, the display screen looks like this:



Then when the EXECUTE key is pressed, statements 50 and 60 read the message (up to 50 characters) from the display screen, and statements 70 and 80 rewrite the message on line 11 and write the new data starting in position 769 (line 3). The display screen looks like this:



Note: You can use the WRITE FILE or REWRITE FILE statement to position the cursor for a following READ FILE statement.

USING PROCEDURE FILES

A procedure file allows you to set up and execute a series of programs without the need for operator intervention. The procedure file consists of a series of BASIC statements, commands, or data created using the LOAD0,DATA command or a BASIC program. After you enter LOAD0,DATA, the system displays a line number followed by a colon. You may then enter statements, commands, or data just as you would for a standard data file. Lines in a procedure file have a maximum length of 64 characters. After you have entered the lines in your procedure file, you can use the SAVE command to save the file on tape or diskette. Following is a sample procedure file:

```
LOAD0,DATA
0010:LOAD 5
0020:RUN
0030:LOAD 7
0040:RUN
0050:LOAD 9
0060:RUN
0070:ALERT INSERT PAYROLL DISKETTE ENTER GO TO CONTINUE
0080:LOAD 11
0090:RUN
```

After these lines are entered, the file can be saved with this SAVE command:

```
SAVE 4,'PROC',RECL=64,D80
```

This command saves the file (named PROC) with a record length of 64 characters in file 4 on diskette drive 1.

The procedure file is accessed when you enter a PROC command. A PROC command instructs the system to begin using the procedure file in the indicated file, as shown below:

```
PROC 4,D80
```

The system then loads file 4 and begins executing the lines in the file, one record at a time. In the example above, after the program in file 9 has been run, the procedure file executes the ALERT command, which sounds an audible alarm and causes the display screen to flash. The operator must press the ATTN key to stop the flashing message INSERT PAYROLL DISKETTE–ENTER GO TO CONTINUE. If the RUN IN=P command was used, data may not be entered from the keyboard. Therefore, for each INPUT statement of a program being executed, there must be one entry available via the procedure file. After the operator enters GO, the procedure file loads file 11. Other commands valid in a procedure file are discussed in the *IBM 5110 BASIC Reference Manual*, SA21-9308.

USING THE SYSTEM CONTROL FUNCTIONS

A unique 35-byte 5110 internal storage area is provided to allow you to dynamically control the operation of your system. This storage area is called file FLS (system file). Using file FLS you can get information about the system, such as: total work area available, work area available for variables and buffer storage, number of lines printed on the printer, and the return code set by the last STOP or END statement. File FLS also allows you to set or use system functions, such as turning the display screen on and off, sounding the audible alarm, selecting a character set, entering lowercase alphabetic characters, turning trace on and off, rounding precision, file FLx, and number of print lines per inch. You use the READ FILE FLS statement to obtain the information about the system, and the WRITE FILE FLS statement to set or use the system functions. (File FLS is always open, therefore, an OPEN statement is *not* required before file FLS is used.

READ FILE FLS Statement

You can obtain the following list of information about the system using the READ FILE FLS statement:

Information	*File FLS Position*
Total work area available	1-5
Work area available for variables and buffer storage	6-10
Number of lines printed	11-15
Reserved	16-18
Return code set by last STOP or END statement	19-21
Unused	22-35

For example, the statements:

```
0320 READFILE USING 330,FLS,A,B,C,D
0330 FORM NC5,NC5,NC5,POS19,NC2
```

place the total work area available in variable A, the work area available for variables and buffer storage in variable B, and so on.

Note: The number of lines printed is the count of the print head movements rather than the form's movements, because a 00 line spacing may be specified.

WRITE FILE FLS Statement

You can set or use certain system functions by using the WRITE FILE FLS statement to place the appropriate code in file FLS, as follows:

System Function	*File FLS Position*	*Code*
Turn the display screen off	1	F
Turn the display screen on	1	N
Turn the audible alarm on	1	S
Turn the audible alarm off	1	Q
Pulse the audible alarm	1	A
Set keyboard input to lowercase character mode	2	L
Set keyboard input to standard BASIC character mode	2	U
Turn the display trace on	3	N
Turn the display trace off	3	F
Turn the printer trace on	4	N
Turn the printer trace off	4	F
Set rounding precision	5-6	0 to 15
File FLx	7-9	FL0-9
Set number of print lines per inch (2.54 centimeters) For example, the statements	10-11	8-99

```
0280 WRITEFILE USING 290,FLS,'L'
0290 FORM POS2,C
```

place the character L in position 2 of file FLS. This causes the 5110 to be in lowercase character mode. That is, lowercase alphabetic characters are entered from the keyboard unless the shift key is used. The 5110 remains in lowercase character mode until the 5110 is changed back to standard BASIC character mode or the work area is cleared.

Additional Use of File FLS

You can use positions 22 through 35 of file FLS to store data. That is, data written to these positions using the WRITE FILE FLS statement can be read using the READ FILE FLS statement. Using these positions of file FLS gives you the unique capability to write data in numeric form and then read that data in character form, and vice versa. For example:

```
0110 INPUT N
0120 WRITEFILE USING 130,FLS,N
0130 FORM POS22,NC8
0140 DIM A$8
0150 READFILE USING 160,FLS,A$
0160 FORM POS22,C8
```

The numeric input (N) is written to file FLS positions 22 through 30 (statements 120 and 130); then the character equivalent of N is read from file FLS positions 22 through 30 (statements 150 and 160).

Using the UTIL Command

You can use the UTIL command to perform the following system control operations:

- List the file directory
- Rename a file
- Change or display a diskette volume ID and owner ID
- Drop a file
- Write-protect a file
- Transfer control to the Diskette Sort feature, if installed
- Change the system default device

The sort program, if installed, is internal to your 5110. You can transfer control to the sort program by using the statement:

```
UTIL SORT
```

This program allows you to sort records in a data file according to specified fields within the records. See the *IBM 5110 Customer Support Functions Reference Manual*, SA21-9311, for a complete description of the sort program.

Normally, the 5110 Model 1 default device is E80 (the built-in tape unit) and the 5110 Model 2 default device is D80 (diskette drive 1). You can use the UTIL command to change the system default device. For example:

```
UTIL SYS D40
```

changes the default device to diskette drive 2.

See the *IBM 5110 BASIC Reference Manual*, SA21-9308, for a complete description of the UTIL command and functions.

An array is a simple way to keep together data items that are related. For example, if you want to keep the average temperature for each month of the year, you could construct an array having 12 data items. The DIM statement can be used to define an array:

```
0010 DIM T(12)
```

This statement defines an arithmetic array, T, containing 12 items, or *members*. The computer recognizes an item as an array by the appearance of parentheses. The parentheses are used to define the number of items in the array.

Arrays can be arithmetic or character. For example:

```
0020 DIM T$18(12)
```

This statement defines a character array having 12 members.

A DIM statement can specify the length of the members of a character array at the same time it is defining the array:

```
0020 DIM T$10(12)
```

Here, each member of array T$ is assigned a length of 10; without the length specification, each member, like other character variables, would be assumed to be 18 characters long. All members of a character array have the same length.

NAMING ARRAYS

Character arrays are named in exactly the same way as character variables; that is, the name must consist of a single alphabetic character (including the alphabet extenders) followed by the currency symbol(s). Thus, the name A$ can name either a character array or a character variable. Arithmetic arrays are named in almost the same way as arithmetic variables. An arithmetic array name may consist *only* of a single alphabetic character (including the alphabet extenders); you may recall that arithmetic variables can also be named with an alphabetic character followed by a digit. Thus, the name A can be used for either an arithmetic array or arithmetic variable, but the name A1 can be used only for an arithmetic variable.

DEFINING ARRAYS

Defining an array in a DIM statement is known as an *explicit* declaration. There is another way to define an array through *implicit* declaration; that is, by referring to a member of an array in a program statement without having defined it first in a DIM statement. When you refer to an array member without explicitly declaring it in the DIM statement, the computer will recognize that you are working with an array and will automatically allow space for 10 members. To refer to a particular member of an array, you specify it by its location in the array. For example, T(1) refers to the first member of the array named T, T(2) refers to the second member, T(3) refers to the third member, and so on. Each number giving the location of a particular member is called a subscript. If the following statement appears in the program:

```
0040 T(9)=69
```

only the ninth member of T would be assigned the value 69; all other members would remain unchanged.

Remember that an array defined implicitly is assumed to have 10 members. So in order for array T to contain 12 members, we must explicitly define it. If an array has very few members (for example, two or three), it would be wise to use a DIM statement, such as:

```
0010 DIM A(2),B(3)
```

The DIM statement, in addition to defining the number of members in the array, also defines the number of *dimensions* in the array.

So far, we have discussed only one-dimensional arrays. In BASIC, you can also have arrays of two dimensions. Assume that values have been assigned to array T, such that:

```
T(1)        is      31
T(2)        is      43
T(3)        is      42
T(4)        is      57
T(5)        is      64
T(6)        is      73
T(7)        is      79
T(8)        is      79
T(9)        is      69
T(10)       is      58
T(11)       is      44
T(12)       is      39
```

Let's assume that these values represent the average temperatures for 12 months; T(1) represents January's average, T(2) February's, and so on.

For various reasons, another programmer might want to consider the year as divided into four quarters of three months each; he could define his array (call it M) as a two-dimensional array, as follows:

```
0015 DIM M(4,3)
```

In this statement, array M is defined as a two-dimensional array containing 12 members (the product of 4 and 3), just like array T. The difference is that the members of M are distributed over two dimensions, whereas in T they are distributed over only one dimension. Conceptually, the two dimensions of M can be thought of as rows and columns–four rows and three columns. The first value would be identified as being in the first row and the first column, or as M(1,1); the second value would be in M(1,2), the first row, the second column; the third in M(1,3), the first row, third column. The fourth item is M(2,1), or the second row, the first column; the fifth item, M(2,2), would be in the second row, second column, and so on.

Assuming that the same temperatures assigned to array T are assigned to array M, notice the difference in the way each item is referred to:

Array T	Temperature	Array M
T(1)	31	M(1,1)
T(2)	43	M(1,2)
T(3)	42	M(1,3)
T(4)	57	M(2,1)
T(5)	64	M(2,2)
T(6)	73	M(2,3)
T(7)	79	M(3,1)
T(8)	79	M(3,2)
T(9)	69	M(3,3)
T(10)	58	M(4,1)
T(11)	44	M(4,2)
T(12)	39	M(4,3)

	Column		
Row	1	2	3
1	31	43	42
2	57	64	73
3	79	79	69
4	58	44	39

Two subscripts are needed to refer to a particular member of array M; for example, M(3,1) refers to the temperature for July, the first month in the third quarter.

Note the difference between a subscript and the array dimension specification. A subscript *refers* to a particular member of an array. It can be any valid arithmetic expression (for example, a numeric constant or an arithmetic variable). The dimension specification *defines* the number of members of an array. The dimension specification can appear only in a DIM statement and it must be indicated by unsigned integers only. An array name cannot appear in a DIM statement if the array has already been defined–either implicitly by usage or explicitly by definition in a previous DIM statement.

You can implicitly define a two-dimensional array by using it in a program statement without defining it in a DIM statement first. You would do this by referring to a particular member, using two subscripts. For example, A(4,3) would refer to the item in the fourth row, the third column of array A. A two-dimensional array defined implicitly will be assigned the dimensions (10,10), or 100 members altogether. If the value of either dimension is to exceed 10, however, you must use a DIM statement to define the array as you would for a one-dimensional array that exceeds 10 members. Remember that DIM statements to define arrays must appear in the program *before* you refer to the array.

PLACING VALUES INTO ARRAYS

Initially the system sets all arithmetic arrays to zero and all character arrays to blanks. Arrays can be given other values through assignment, READ, and INPUT statements, just like other variables. The assignment statement can assign values to individual array members or to all the members of the array. Here are some examples:

```
0300 A(4,5)=10
0320 MAT A=(15)
0330 P$(4)='PHILADELPHIA'
```

The first example assigns the value 10 to the member in the fourth row, fifth column of the two-dimensional arithmetic array A. In the second example, the keyword MAT identifies A as an array, and the value 15 is assigned to all the members of the array. (This is a special form of the assignment statement and is known as the array assignment statement.) In the third example, the value PHILADELPHIA is put into the fourth member of the one-dimensional character array P$.

When specifying values by means of READ and INPUT statements, you must remember that every array member that is to receive a value must be represented in the statement, and a value must be supplied for each member specified. Let's look at these statements:

```
0010 DIM T(12),T$18(12)
0015 PRINT 'ENTER 3 TEMPERATURES, THEN THREE MONTHS'
0020 INPUT T(1),T(2),T(3),T$(1),T$(2),T$(3)
```

The DIM statement defines the arithmetic array T and the character array T$, each with 12 members. The INPUT statement states that values will be supplied at execution time for the first three members of each array. Execution of the INPUT statement causes the computer to display the question mark (?). A valid response would be:

```
31,43,42,JANUARY,FEBRUARY,MARCH
```

The first three values are entered into T(1), T(2), and T(3), respectively. The next three values are entered into T$(1), T$(2), and T$(3), respectively.

The following statement can be used to enter values for the arithmetic array A, consisting of three rows and four columns:

```
0020 DIM A(3,4)
0030 MAT INPUT A
```

When this statement is executed, the computer displays a question mark, and you can enter the three values for the first row:

The system continues to display the question mark until you have entered values for all matrix positions.

Another way of assigning input values to arrays is through use of a FOR/NEXT group in conjunction with the READ and DATA statements. For example, if you wanted a list of 15 numbers assigned to an array named B, you could write:

```
0010 DIM B(15)
0020 FOR I=1 TO 15
0030 READ B(I)
0040 NEXT I
0050 DATA 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

The subscript I is used to step through the values in the data table.

REDIMENSIONING ARRAYS

Once an array has been dimensioned by a DIM statement, it cannot be explicitly dimensioned again. But it can be *redimensioned*; that is, the array can be given new dimensions. A one-dimensional array can be redimensioned into a two-dimensional array, or it can be redimensioned into a one-dimensional array with a different number of members. Similarly, a two-dimensional array can be redimensioned into a one-dimensional array or a two-dimensional array having a different number of members in either or both dimensions. *The rule to remember when redimensioning an array is that the total number of members in the new array may not exceed the total number in the original array.* For example, the array M(12,10) has 120 members, the product of 12 and 10. It can be redimensioned as long as the new array does not contain more than 120 members (it can contain fewer). Thus, M(12,10) may be correctly redimensioned to M(40,3), or M(100), but not to M(40,4).

One way to redimension an array is to state its new dimensions right after the array name in the array assignment statements. For example, in the array C(5,5) to C(3,4), you could use the array assignment statement:

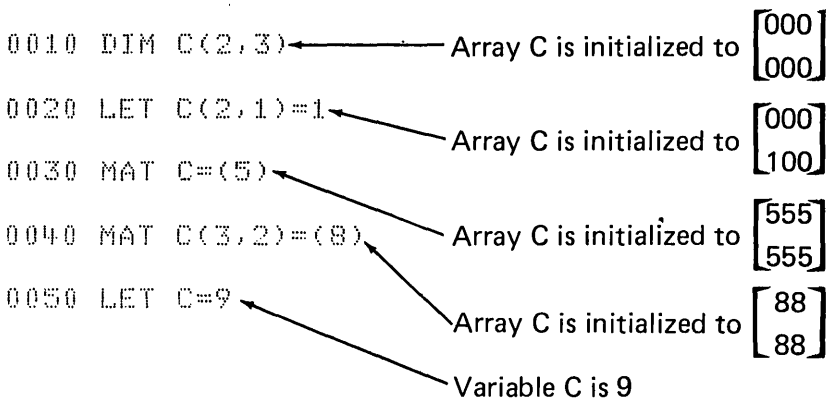
```
0010 MAT C(3,4)=(0)
```

The word MAT is used to indicate that operations are to be performed on the entire array, or matrix. This statement changes the array dimensions to (3,4) and assigns the value zero to each member of the newly dimensioned array.

DIFFERENCE BETWEEN MAT AND LET

It is important to note the distinction between the array assignment statement, identified by the word MAT, and the LET assignment statement.

The following example shows a sequence of assignment statements and the output from each one. None of the statements are equivalent.



Statement 10 defines arithmetic array C as a 2x3 array and initializes each member to 0. Statement 20 assigns the value 1 to a member in the second row, first column of the array. Statement 30 assigns the value 5 to all members of the array. Statement 40 redimensions the 2x3 array into a 3x2 array and assigns the value 8 to all members. Statement 50 does not refer to an array but to an arithmetic variable, C, and assigns the value 9 to it. BASIC allows you to use the same name to represent both an array and a simple variable in the same program.

The array assignment statement can also assign the values of an array to another array, as long as both arrays have identical dimensions. Let's look at this example:

```
0100 DIM Y(4),Z(4)
  .
  .
  .
0150 MAT Y = (A*B)
  .
  .
  .
0180 LET Y(3)=15
  .
  .
  .
0200 MAT Z=Y
```

Statement 150 assigns the value of the expression A*B to all the members of the array Y. The expression must always be enclosed in parentheses. Statement 180 assigns the value 15 to the third member of Y. Note the difference between the LET statement and the MAT statement. Statement 200 assigns the values in array Y to array Z. If the only change made to array Y between statements 150 and 200 was the assignment made in statement 180, array Y will contain the values A*B in members 1, 2, and 4 and the value 15 in member 3. Array Z will be assigned these values in the corresponding members.

In order for the values of one array to be assigned to another, both arrays must have identical dimensions. For example, if Z had the dimension (5) or (2,2), it would have to be redimensioned to the dimensions of Y before it could receive Y's values.

ARRAY OPERATIONS

A number of different operations can be performed on arrays. Arithmetic arrays can be used in simple arithmetic operations, such as adding or subtracting the values of members in different arrays, and in true mathematical matrix operations such as matrix multiplication. Additionally, values in both arithmetic and character arrays can be indexed in ascending or descending order. Arrays used in arithmetic operations must have the same number of dimensions. Let's look at some of the operations available.

Array Addition and Subtraction

Example 1:

```
0010 DIM X(5),Y(5),Z(5)
0020 MAT X=Y+Z
```

In this example, each member of the array X is to be assigned the sum of the corresponding members of the arrays Y and Z. The values of Y(1) and Z(1) are added, and the sum is assigned to X(1); the values of Y(2) and Z(2) are added and assigned to X(2), and so on.

Example 2:

```
0030 DIM X(5),Y(5),Z(5)
0040 MAT X=Y-Z
```

This example is like the first example, except that the array X is assigned the difference between the corresponding members of the arrays Y and Z.

Scalar Multiplication

Scalar multiplication is the process whereby each member of an array is multiplied by the same number.

Example:

```
0035 DIM A(10,5),B(14)
0040 MAT A(14)=(4)*B
```

In statement 40, A is redimensioned to correspond to the dimensions of the array B. Then, the value in each member of B is multiplied by 4 and the product is assigned to the corresponding member of A; B(1)*4 is assigned to A(1), B(2)*4 to A(2), and so on.

Indexing Function

Indexing operations can be performed on character as well as on arithmetic arrays. Character arrays are indexed alphabetically, arithmetic arrays numerically. Arrays can be indexed in ascending or descending sequence by the AIDX and DIDX functions, respectively.

Example:

```
0010 DIM A$18(5),B(5)
0020 DATA 'DAN','MEL','GLEN','DAVE','BILL'
0040 MAT READ A$
0050 MAT PRINT FLP,A$
0060 MAT B=AIDX(A$)
0070 MAT PRINT FLP,B
```

The printed output would be:

```
DAN           MEL           GLEN          DAVE          BILL


 5             1             4             3             2
```

The numbers indicate the ascending character sequence of the names entered according to the order in which they were entered. For example, the 5 indicates that the fifth name entered (BILL) is the lowest character value entered, the 1 indicates that the first value entered (DAN) is the next lowest, and so on.

The following statements could be added to print the indexed matrix:

```
0080 FOR I=1 TO 5
0090 PRINT FLP,A$(B(I))
0100 NEXT I
```

The printed output would be:

```
BILL
DAN
DAVE
GLEN
MEL
```

Matrix Multiplication

Matrix multiplication is the process whereby the matrix product of two arithmetic arrays is assigned to a third array. All three arrays involved in matrix multiplication must be two-dimensional.

Example 1:

```
0065 DIM X(2,2),Y(2,2),Z(2,2)
0070 MAT Z=X*Y
```

If X contained $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and Y contained $\begin{bmatrix} e & f \\ g & h \end{bmatrix}$

the values of Z $\begin{bmatrix} j & k \\ l & m \end{bmatrix}$, would be constructed as follows:

j = a*e + b*g
(sum of members in first row of X times members in first column of Y)

k = a*f + b*h
(sum of members in first row of X times members in second column of Y)

l = c*e + d*g
(sum of members in second row of X times members in first column of Y)

m = c*f + d*h
(sum of members in second row of X times members in second column of Y)

All the arrays shown in example 1 are two-dimensional, square, and have the same number of members. Arrays used in matrix multiplication need not be square or have the same number of members, but must be two-dimensional and *comformable*. Look at this example:

```
0075 DIM A(2,4),B(4,3),C(2,3)
0080 MAT C=A*B
```

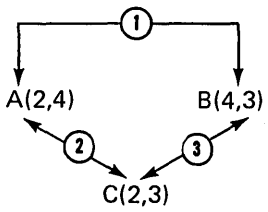
Remember that the first subscript in a two-dimensional array indicates the number of rows, and the second subscript indicates the number of columns. (In the example above, A has two rows and four columns.) To be conformable for matrix multiplication, arrays must meet these requirements:

- The number of columns in the first array to be multiplied must equal the number of rows in the second. In the example above, A(x,4)= B(4,x).

- The number of rows in the receiving array must equal the number of rows in the first array. In the example, C(2,x)=A(2,x).

- The number of columns in the receiving array must equal the number of columns in the second array. In the example, C(x,3)= B(x,3).

These requirements are graphically represented below:



The arrays in statements 75 and 80 are conformable and thus are valid for matrix multiplication operations.

If A contained $\begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix}$ and B contained $\begin{bmatrix} i & j & k \\ l & m & n \\ o & p & q \\ r & s & t \end{bmatrix}$

the values of C $\begin{bmatrix} u & v & w \\ x & y & z \end{bmatrix}$, would be constructed as follows:

u = a*i + b*1 + c*o + d*r
(sum of members in first row of A times the members in first column of B)

v = a*j + b*m + c*p + d*s
(sum of members in first row of A times the members in second column of B)

w = a*k + b*n + c*q + d*t
(sum of members in first row of A times the members in third column of B)

x = e*i + f*l + g*o + h*r
(sum of members in second row of A times the members in first column of B)

y = e*j + f*m + g*p + h*s
(sum of members in second row of A times the members in second column of B)

z = e*k + f*n + g*q + h*t
(sum of members in second row of A times the members in third column of B)

Chapter 14. What to Do When Your Program Does Not Work

When your program does not work properly, you can use the following 5110 aids to assist you in determining what is wrong:

- Program trace
- Program step
- Comments
- Keyboard-generated data files

PROGRAM TRACE

Program trace allows you to trace the order in which program statements are executed. Each statement number is displayed (and printed if you specify PRINT with the RUN or GO command) as the statement is executed. The following example shows the display and printout when the RUN TRACE, PRINT command is entered.

Sample program:

```
0010 DIM Q(5)
0020 PRINT 'ENTER 5 TEMPERATURE QUOTATIONS'
0030 MAT INPUT Q
0040 FOR I=1 TO 5
0050 T=T+Q(I)
0060 NEXT I
0070 A=T/5
0080 PRINT '5 DAY MOVING AVERAGE =';A
0090 STOP
```

The display shows:

```
RUN TRACE,PRINT
0010 0020 ENTER 5 TEMPERATURE QUOTATIONS
0030
62,65,68,61,64
0040 0050 0060 0050 0060 0050 0060 0050 0060 0050 0060 0070
0080 5 DAY MOVING AVERAGE = 64
0090
```

The printed output is:

```
0010 0020 0030 0040 0050 0060 0050 0060 0050 0060 0050 0060
0050 0060 0070 0080 0090
```

Your program could stop because an error occurred, or you could stop execution by inserting PAUSE statement(s) in your program; for example:

```
0035 PAUSE
0040 FOR I=1 TO 5
0050 T=T+Q(I)
0055 PAUSE
0060 NEXT I
```

The PAUSE statements allow you to trace and analyze just the part of the program that is not working correctly. When the program above pauses at statement 0035, you can start it again by entering GO 40, TRACE. The program then pauses at statement 0055. While the program is halted for the PAUSE statement, you can check the value of variables to see if your program is progressing properly. See the sample cross-reference program in Chapter 15, *Tips and Techniques* for a method of listing variables to determine where they are used and whether they are used more than once.

You can also start and stop a program trace during program execution using the WRITE FILE and FORM statements. For example:

```
6255 WRITEFILE USING 6260,FLS,'N','N'
6260 FORM POS3,C,POS4,C
6300 WRITEFILE USING 6260,FLS,'F','F'
```

Statement 6255 turns on trace with output to both the display (N in position 3 of file FLS) and the printer (N in position 4 of file FLS). Trace remains on until statement 6300 when the WRITE FILE statement turns it off (writes an F in positions 3 and 4 of file FLS).

PROGRAM STEP

With program step, you can execute a program one step at a time, which can be helpful when you analyze complex routines. As with trace, you can execute part of the program in step mode by inserting a PAUSE statement at the beginning of the routine (or statements) you want to analyze.

For example:

```
0010 DIM Q(5)
0020 PRINT 'ENTER 5 TEMPERATURE QUOTATIONS'
0030 MAT INPUT Q
0035 PAUSE <-------- Allows you to start program step
0040 FOR I=1 TO 5
0050 T=T+Q(I)
0060 NEXT I
0065 PAUSE <-------- Allows you to stop program step and/or
                     analyze program results.
0070 A=T/5
0080 PRINT '5 DAY MOVING AVERAGE = ';A
0090 STOP
```

When the program pauses at statement 0035, entering G0 40, STEP causes the program to execute one statement at a time, allowing you to check program results. For example:

```
RUN
ENTER 5 TEMPERATURE QUOTATIONS

62,65,68,61,64
G040,STEP <-------- Begin program step at statement 40.

T,I <-------- Request the values of variables T and I.
 62                1
T,I
 320               5

G070,RUN <-------- Continue processing at statement 70 without
                   program step.
5 DAY MOVING AVERAGE =  64
```

COMMENTS

Using comments within your program can help you remember program logic and aid in analyzing program problems. When you are finished developing your program, you can remove the comments or revise them for future program analysis. Comments use storage and a small amount of execution time. Thus, you should use comments carefully if you are concerned with performance or storage use. However, comments can be an important aid in future analysis of your program, especially if someone else must maintain the program.

Keyboard Test Data Files

When developing or analyzing a program, you might have to use test data. You can use keyboard test data file(s) to create a test file on the display screen. You can open the screen for both input and output and for both stream I/O and record I/O files. For example, you can open line one of the screen as a stream I/O input file as shown:

```
0020 OPEN FL3,'001',IN
```

References to file FL3 imply that data is to be entered from the keyboard; for example:

```
0360 GET FL3,A$,B,C
```

This statement indicates that an alphabetic field followed by two numeric fields will be read from the file referenced by FL3; for example:

```
Allen Brown, 4.80, 6085.56
```

You could also use lines 1 through 14 as a record I/O file, for example:

```
0020 OPEN FILE FL2,'002',ALL
0400 READFILE USING 410,FL2,A$,B,C
0410 FORM POS1,C20,NC5,NC3
```

References to file FL2 indicate that data is to be entered from the keyboard.

You can enter test data as necessary to thoroughly test your program during program development. When you are finished testing, you can change device addresses to the value you will use in your finished program.

Chapter 15. Tips and Techniques

Often, specific examples can aid you in understanding the operation of a function or a group of functions. This chapter shows examples and describes different techniques that you may find helpful in developing and using your programs. The topics included in this chapter are:

- Performance considerations
- Storage considerations
- Program analysis using a cross-reference program
- Skipping to a new page while printing
- Locating a character in a string
- Testing for an error
- Sorting an index file
- Another way to read a stream I/O file
- Examples of the different file access methods

PERFORMANCE CONSIDERATIONS

As you optimize the performance of an application, you may want to consider the following:

Program design
Index file sorting
Print overlap
Display off
Main storage index area
Data file access selection

Program Design

Performance of an application is enhanced if it is initially designed carefully and thoughtfully. Flow diagrams are very helpful in designing efficient running systems. There are many publications on flowcharting that you may find helpful if you are not familiar with the technique.

Index File Sorting

Many applications, such as inventory, make use of an index file with pointers that allow fast access to desired records. If the index file for the inventory example is sorted in ascending order, access to master inventory records will be faster. The increased performance occurs as the result of the fast scan feature implemented in the 5110, which requires a sorted file. As new items are added to the master file, the item number key (item number is specified as the key) is added to the end of the index file. Depending on the activity of adding and deleting records, the index file should be periodically sorted so that the new index record is placed in its proper location and the unwanted index records are deleted. You can sort the index file using the 5110 Diskette Sort feature; see *Sorting an Index File* in this chapter.

Print Overlap

The 5110 can overlap printer output with computer processing. If it is possible with your application, the printed output might be as illustrated below:

```
0010 CALCULATION
      .
      .
      .
0050 PRINT FLP
0060 CALCULATION
      .
      .
      .
0100 PRINT FLP
0110 CALCULATION
      .
      .
      .
```

In the above illustration, calculations to be included in the next print statement are being performed while the previous line is being printed.

Display Off

Some applications may require extensive periods of processing time. Such applications should execute faster if the display screen is turned off so that the 5110 does not have to take the time to keep the display generated. You can turn off the display screen by writing an F in position 1 of file FLS. This procedure is described under *Using the System Control functions* in Chapter 12.

Main Storage Index Area

Access to a master file record using an index file can be improved substantially if you maintain a main storage index area that points to the index file. To do this, you use the KW= parameter, which is included in the OPEN statement for the index file. For example,

```
30 OPEN FILE FL2, 'D80', 9, 'TAXES', IN, KEY, KW=900
```

In the above statement, 900 bytes of main storage have been allocated for index file pointers. Use of this storage area can best be described with an example.

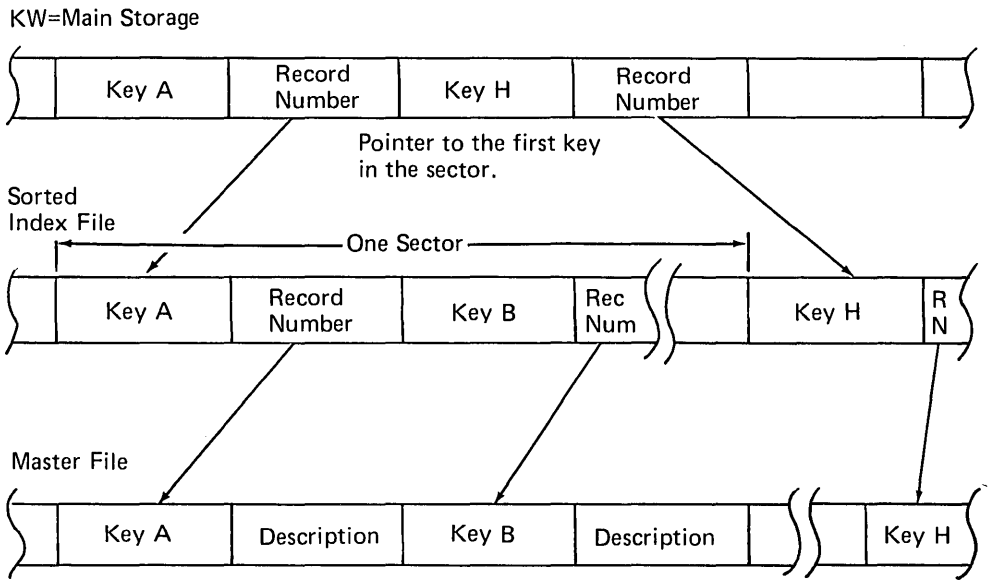
Consider an inventory application with the following characteristics:

- The maximum number of items in the master file is 1000 items.
- The key to the master file is the item number which is 12 bytes.
- The diskette format is 256 bytes per sector.
- The master file record size is 100 bytes per record.

The following questions can be asked:

- How large should the index file be?
- How large should the storage index area be?

Before you answer the above questions, study the following diagram to help you understand the use of the KW storage area and index file.

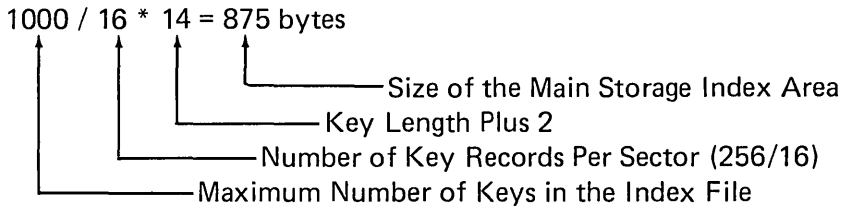


The index file, maintained in sorted order, contains each key and the record number of each record in the master file. A key record is always 8, 16, or 32 bytes. In the example, the key length (item number) is 12 bytes. A master file record number is 4 bytes, giving 16 bytes total for each key record.

The main storage index area contains the first key in a sector and the physical record location of the key in the index file.

The index file sector containing the item number key is found by comparing the item number to the keys in storage. Because the index file and main storage index area are in sorted order, the sector location of the key index record can be quickly found. The system proceeds to the sector designated and reads the sector sequentially until it finds the matching key. After the matching key is found, the master file address is read and used to directly access the item master record. If no key match was found in the index sector, the system proceeds to the end of the file to see if new records have been added. If no key match occurs at the end of the index file, an error occurs.

Now, to answer the first question, the index file size is found by multiplying the maximum number of keys by the key length, which is 1,000*16 or 16K. The size required for the storage index area is calculated as follows:



This example, using KW=900 is slightly greater than the exact amount of storage area to contain one key for every sector in the index file.

The above procedure produces the most efficient method of accessing the master file by index key. However, you need not have one key in storage for every sector in the index file. If storage is limited, as little as one key in storage (KW=14) would improve access time by starting the search in the middle of the index file as required.

Data File Access Selection

One of the most important decisions is choosing the proper access method for your data files.

Whether to use the sequential, direct, or indexed access method depends on your application.

Individual Record Access

The fastest method of access to an individual record is directly by means of the relative record number of the desired record.

For example, in an inventory file it is possible to convert the item number into a record number. Item numbers could be 1 to 1000. Item number 52 would be record 52 in the file. There are more complicated methods for creating a relative record number; however, they are beyond the scope of this document.

Indexing is the next fastest method to access individual records. A pointer to the master file data record is maintained in an index file. This is the most commonly used access method because existing keys such as item numbers can be used without chance of duplicates.

Processing a file sequentially to find an individual record is time consuming because the file must be read from the beginning until the proper record is found.

Sequential Access

If a file can be accessed sequentially, the fastest method would be to sort the master file into the desired order before processing. If the file is processed sequentially in some cases and directly in others, it may be more appropriate to create a sorted index file. The system can then access the master file sequentially by accessing the index file sequentially or directly by providing a key to the index file.

Storage Considerations

User Storage

The amount of user storage available to you for application programs depends upon your 5110 model. Four different storage sizes are available:

16K bytes
32K bytes
48K bytes
64K bytes

Any model can be up-graded to the next higher model by the addition of 16K of storage. In all models, approximately 4K bytes are used for system-related functions. The remaining storage is available for program and data storage. It is a good idea to subtract a buffer of 1K bytes when estimating storage requirements.

Considering a 32K machine, for example, you would subtract 5K, leaving 27K bytes for your program and data. The amount of storage used for a program is a function of many items:

- Program Design
  - Variables
  - Program statements
  - Buffers
  - Precision

Careful control of the above items should lead to both smaller programs and more efficient programs.

Program Design

Storage is allocated for each program statement you write and each variable you use. Careful program design should eliminate unnecessary program statements and variables. A flow diagram prepared for each essential step of the application will aid you in writing the program. Commonly used calculations, such as tax calculation, can be quickly identified and written as a subroutine rather than rewritten in various parts of your programs. Your application may lend itself to being divided into individual programs, each with a specific function.

- Application

  - Data entry
  - Data edit/update
  - Sort
  - Process/update
  - Print reports

The above functions could describe the steps in many different applications. Each of these may, perhaps, also be subdivided into smaller programs.

Addressing the elements of the application one at a time, rather than attempting to write the entire application as a single program, should result in easier programs to write and understand, and require less storage for execution.

Variables

Each time a new variable is used in your program the system automatically assigns a predetermined (default) amount of storage for the data in that variable. For example:

```
0020 A$='JAMES SMITH'
```

The character variable A$ was assigned 18 character positions in storage even though the data 'JAMES SMITH' occupies only 11 positions.

If the data in A$ is constant or can be limited to 11 positions, you can use a dimension statement to override the default value and assign only the necessary 11 positions to A$, thus using only the amount of storage absolutely necessary. For example:

```
0010 DIM A$11
```

A specific amount of storage is required for the definition of each variable as it is encountered; this amount does not include the space allocated to that variable for data storage. In the above example, with *no* dimension statement, 4 bytes are required for the A$ definition, bringing the storage utilization to 18+4 or 22 bytes. The amount of storage required for the different variable definitions and data storage is specified in the *IBM 5110 BASIC Reference Manual*.

Use of matrix variable definition can also help in conserving storage. Suppose four character-fields are to be used as follows:

A$	Name
B$	Street Address
C$	City
D$	State

Assuming the data storage for each variable defaults to 18 characters, a total of 88 bytes of storage would be required. If the same data were placed in a four-element matrix, the amount of storage used would be 4 elements * 18 bytes of data plus 10 bytes for the matrix definition. For example:

(4 * 18)+10=82 bytes

Program Statements

Program statements also occupy storage; this is a more difficult item to estimate due to the complexities of each statement. As a rule of thumb, the approximate amount of storage required for program statements can be calculated by multiplying the number of program statements times the average number of characters (excluding delimiter blanks) per statement.

Program Statement	Number of Characters
`0010 PRINT 'ENTER PRICE'`	22
`0020 INPUT P`	10
`0030 PRINT 'ENTER QTY'`	20
`0040 INPUT Q`	10
`0050 T=P*Q`	9
`0060 PRINT 'TOTAL COST ';T`	26
`0070 GOTO 0010`	10

In the above example, there is an average of approximately 16 characters per line. The storage estimate for the program statements is 7 * 16 or 112 bytes.

The actual amount of user storage available is displayed in the lower right corner of the display when the 5110 is in the ready state. For a 64K system this is 65,536-4,624 or 60,912. The 4,624 bytes represent system work space.

A more accurate method to determine program statement storage is to save the program on tape or diskette. When the program is reloaded, the amount of user space left will be displayed in the lower right corner. Subtracting this number from 60,912 yields the actual program statement requirements.

The storage requirement for the example above is 60,912-60,780 or 132 bytes.

After execution the storage available is 60,754 bytes, indicating that 26 bytes were assigned to variables and data when the program was run.

Buffers

Buffer storage is required for operation involving data files, printer output, and the special function using A$.

Data Files

Each time a stream I/O data file is opened, a storage buffer of 50 bytes plus the physical record length is allocated. The physical record length is 512 bytes if the file references tape, and the physical record length is the sector size if the file references the diskette.

Each time a record I/O file is opened a storage buffer of 68 bytes plus a multiple of the physical record length is allocated. Most commonly the multiple is 1 or 2. The physical record length for tape is always 512 bytes. For diskette it can be 128, 256, 512, 1024 depending on how the diskette was initialized. Record I/O buffers are discussed in the *IBM 5110 BASIC Reference Manual*.

Printer

Printer output requires a buffer storage of 200 bytes.

Using A$

The using A$ parameter is used with the READ and PRINT statements

```
0020 A$='FORM 3*NC5'
0340 PRINT USING A$,FLP,A,B,C
```

The first time a using A$ parameter is encountered, the 5110 automatically allocates a 420-byte buffer, which is used by all subsequent using A$ statements. Statements referencing other variable identification (such as B$) also use the same buffer area.

Precision Long and Short

When a program is in execution, each numeric variable is carried in long precision (fifteen digits) and occupies eight character positions. By entering RUNS numeric variables are carried in short precision (seven digits) and occupy four character positions. Whether RUNS (short precision) is acceptable depends on the requirements of each individual application.

PROGRAM ANALYSIS USING A CROSS-REFERENCE PROGRAM

Occasionally, while writing a BASIC program with many loops, subroutines, and other functions, you may find that normal debugging techniques are unsuited due to the complexity of the program. The following is a cross-reference program that can be used to cross-reference the occurrence of variables, line numbers, functions, and so on, within any program saved in a file. To do this, simply load the cross-reference program, and respond to its prompting messages for the device address, number, and name of the file containing the program to be cross-referenced. The program to be processed must have been saved in source format with 64 or 128 character record length. For example:

```
SAVE 1,'NAME',SOURCE,RECL=64,D80
```

For details about any of the statements in the cross-reference program, see the *5110 BASIC Reference Manual*.

The following is a listing of the cross-reference program.

```
0010 REM BASIC CROSS REFERENCE PROGRAM - REFERENCE EXTRACT
0020 REM
0030 DIM N$4,B$4,C$39
0040 DIM R$4(3000),S$4(3000),X(3000)
0050 C$='ABCDEFGHIJKLMNOPQRSTUVWXYZ$#@0123456789'
0060 PRINT 'ENTER DEVICE CODE, FILE NUMBER AND FILE ID FOR PGM'
0070 INPUT D$,F,F$
0080 ONERROR GOTO 200
0090 OPEN FILE FL1,D$,F,F$,IN
0100 ONERROR SYSTEM
0110 R=RLN('FL1')
0120 IF R≠64 GOTO 150
0130 DIM A$60,M$59
0140 GOTO 290
0150 IF R≠128 GOTO 180
0160 DIM A$124,M$123
0170 GOTO 290
0180 PRINT 'RECORD LENGTH OF INPUT FILE NOT 64 OR 128'
0190 STOP
0200 IF &ERR≠608 GOTO 270
0210 ONERROR SYSTEM
0220 OPEN FL1,D$,F,F$,IN
0230 WRITEFILE FLS,'      FL1'
0240 T1=1
0250 DIM A$128,M$127
0260 GOTO 290
0270 PRINT 'ERROR DURING OPEN -'&ERR
0280 STOP
0290 WRITEFILE FLS,'F'
0300 REM
0310 REM START OF LOOP TO PROCESS INPUT RECORDS
```

```
0320 REM
0330 GOTO 370 ON T1
0340 READFILE USING 350,FL1,N$,M$,EOF 710
0350 FORM C4,X1,C
0360 GOTO 400
0370 GET FL1,A$,EOF 710
0380 N$=STR(A$,1,4)
0390 M$=STR(A$,6)
0400 PRINT FLP,N$' 'M$
0410 GOSUB 970
0420 IF STR(M$,1,3)='REM' GOTO 330
0430 IF STR(M$,1,1)=':' GOTO 330
0440 IF STR(M$,1,4)='DATA' GOTO 330
0450 A$=M$
0460 REM REPLACE ALL OPERATORS WITH BLANKS
0470 REM
0480 FOR I=1 TO LEN(A$)
0490 IF IDX(C$,STR(A$,I,1))≠0 GOTO 560
0500 IF STR(A$,I,1)≠'''' GOTO 550
0510 J=IDX(STR(A$,I+1),'''')
0520 STR(A$,I,J+1)=' '
0530 I=I+J
0540 GOTO 560
0550 STR(A$,I,1)=' '
0560 NEXT I
0570 REM INPUT RECORD HAS BEEN MODIFIED - EXTRACT REFERENCES
0580 L=LEN(A$)
0590 I=0
0600 I=I+1
0610 IF I>L GOTO 330
0620 IF STR(A$,I,1)=' ' GOTO 600
0630 J=IDX(STR(A$,I),' ')
0640 B$=STR(A$,I,J-1)
0650 I=I+J-1
0660 X1=X1+1
0670 S$(X1)=N$
0680 R$(X1)=B$
0690 IF I≤L GOTO 620
0700 GOTO 330
0710 REM END OF PROGRAM - SORT AND PRINT OUT
0720 GOSUB 1040
0730 MAT R$(X1)=R$
0740 MAT S$(X1)=S$
0750 MAT X(X1)=(0)
0760 MAT X=AIDX(R$)
0770 C0=0
0780 FOR I=1 TO X1
0790 IF N$=R$(X(I)) GOTO 850
0800 GOSUB 970
0810 PRINT FLP,TAB(1),R$(X(I));TAB(5);':';
0820 N=N+1
0830 C0=0
```

```
0840 N$=R$(X(I))
0850 C0=C0+1
0860 IF C0≤10 GOTO 900
0870 GOSUB 970
0880 PRINT FLP,TAB(5),':';
0890 C0=1
0900 PRINT FLP,'   ';S$(X(I));
0910 NEXT I
0920 PRINT FLP
0930 PRINT FLP,'NUMBER OF SYMBOLS     ='N
0940 PRINT FLP,'NUMBER OF REFERENCES ='X1
0950 GOSUB 1040
0960 STOP
0970 READFILE USING 980,FLS,L0
0980 FORM POS11,NC5
0990 L0=INT(L0/66)*66+66-L0
1000 IF L0≤0⁞L0>6 GOTO 1030
1010 PRINT USING 1020,FLP,' '
1020 FORM C1,SKIPL0
1030 RETURN
1040 READFILE USING 1050,FLS,L0
1050 FORM POS11,NC5
1060 L0=INT(L0/66)*66+66-L0
1070 PRINT USING 1080,FLP,' '
1080 FORM C1,SKIPL0
1090 RETURN
```

The following is a sample printout when the cross-reference program is run on itself.

```
A$   :  0370  0380  0390  0450  0480  0490  0500  0510  0520  0550
     :  0580  0620  0630  0640
A$12:  0160  0250
A$60:  0130
AIDX:  0760
B$   :  0640  0680
B$4  :  0030
C    :  0350
C$   :  0050  0490
C$39:  0030
C0   :  0770  0830  0850  0850  0860  0890
C1   :  1020  1080
C4   :  0350
D$   :  0070  0090  0220
DIM  :  0030  0040  0130  0160  0250
EOF  :  0340  0370
ERR  :  0200  0270
F    :  0070  0090  0220
F$   :  0070  0090  0220
FILE:  0090
FLP  :  0400  0810  0880  0900  0920  0930  0940  1010  1070
FLS  :  0230  0290  0970  1040
FL1  :  0090  0220  0340  0370
FOR  :  0480  0780
```

```
FORM:  0350  0980  1020  1050  1080
GET :  0370
GOSU:  0410  0720  0800  0870  0950
GOTO:  0080  0120  0140  0150  0170  0200  0260  0330  0360  0420
     :  0430  0440  0490  0500  0540  0610  0620  0690  0700  0790
     :  0860  1000
I   :  0480  0490  0500  0510  0520  0530  0530  0550  0560  0590
     :  0600  0600  0610  0620  0630  0640  0650  0650  0690  0780
     :  0790  0810  0840  0900  0910
IDX :  0490  0510  0630
IF  :  0120  0150  0200  0420  0430  0440  0490  0500  0610  0620
     :  0690  0790  0860  1000
IN  :  0090  0220
INPU:  0070
INT :  0990  1060
J   :  0510  0520  0530  0630  0640  0650
L   :  0580  0610  0690
LEN :  0480  0580
LO  :  0970  0990  0990  0990  1000  1000  1040  1060  1060  1060
M$  :  0340  0390  0400  0420  0430  0440  0450
M$12:  0160  0250
M$59:  0130
MAT :  0730  0740  0750  0760
N   :  0820  0820  0930
N$  :  0340  0380  0400  0670  0790  0840
N$4 :  0030
NC5 :  0980  1050
NEXT:  0560  0910
ON  :  0330
ONER:  0080  0100  0210
OPEN:  0090  0220
POS1:  0980  1050
PRIN:  0060  0180  0270  0400  0810  0880  0900  0920  0930  0940
     :  1010  1070
R   :  0110  0120  0150
R$  :  0680  0730  0730  0760  0790  0810  0840
R$4 :  0040
READ:  0340  0970  1040
RETU:  1030  1090
RLN :  0110
S$  :  0670  0740  0740  0900
S$4 :  0040
SKIP:  1020  1080
STOP:  0190  0280  0960
STR :  0380  0390  0420  0430  0440  0490  0500  0510  0520  0550
     :  0620  0630  0640
SYST:  0100  0210
TAB :  0810  0810  0880
TO  :  0480  0780
T1  :  0240  0330
USIN:  0340  0970  1010  1040  1070
WRIT:  0230  0290
```

```
X     :  0040  0750  0760  0790  0810  0840  0900
X1    :  0350  0660  0660  0670  0680  0730  0740  0750  0780  0940
0     :  0490  0590  0750  0770  0830  1000
1     :  0240  0380  0420  0430  0430  0440  0480  0490  0500  0510
      :  0520  0550  0600  0620  0640  0650  0660  0780  0810  0820
      :  0850  0890
10    :  0860
1020  :  1010
1030  :  1000
1040  :  0720  0950
1050  :  1040
1080  :  1070
128   :  0150
150   :  0120
180   :  0150
200   :  0080
270   :  0200
290   :  0140  0170  0260
3     :  0420
3000  :  0040  0040  0040
330   :  0420  0430  0440  0610  0700
350   :  0340
370   :  0330
4     :  0380  0440
400   :  0360
5     :  0810  0880
550   :  0500
560   :  0490  0540
6     :  0390  1000
600   :  0620
608   :  0200
620   :  0690
64    :  0120
66    :  0990  0990  0990  1060  1060  1060
710   :  0340  0370
850   :  0790
900   :  0860
970   :  0410  0800  0870
980   :  0970
NUMBER OF SYMBOLS     = 105
NUMBER OF REFERENCES = 376
```

SKIPPING TO A NEW PAGE WHILE PRINTING

Using File FLS

In the cross-reference program (see *Program Analysis Using a Cross-Reference Program*, this chapter), lines 760 to 880 show two methods of skipping to a new page while printing.

- Skip to a new page with 6 or fewer lines left
- Skip unconditionally to a new page.

Both methods use a portion of the contents of file FLS. Positions 11 through 15 of file FLS always contain the total number of lines printed.

```
0970 READFILE USING 980,FLS,L0
0980 FORM POS11,NC5
0990 L0=INT(L0/66)*66+66-L0
1000 IF L0≤0!L0>6 GOTO 1030
1010 PRINT USING 1020,FLP,' '
1020 FORM C1,SKIPL0
1030 RETURN
1040 READFILE USING 1050,FLS,L0
1050 FORM POS11,NC5
1060 L0=INT(L0/66)*66+66-L0
1070 PRINT USING 1080,FLP,' '
1080 FORM C1,SKIPL0
1090 RETURN
```

Statements 760 and 830 are READ FILE statements that access file FLS. Statements 770 and 840 are FORM statements specifying that only the 5 numeric characters beginning in position 11 of file FLS be accessed. From this point on, the two methods of page skipping differ. The subroutine consisting of lines 760 to 820 specifies that printing begin on a new page if space for 6 or fewer lines remains on the current page. The subroutine consisting of lines 830 to 880 specifies that printing begin on a new page unconditionally. In both cases, the constant 66 (11-inch paper at 6 lines per inch) is used as the page length. The calculations in statements 780 and 850 use the data from file FLS (named L0) to determine remaining space on the current page. The IF statement (790) specifies the conditions for skipping to a new page (if space remaining is less than zero or greater than 6, continue printing). Statements 800 and 860 specify that blank lines be printed according to the FORM statements in 810 and 870. These FORM statements also indicate that L0 is the number of lines to be skipped. The following examples show a breakdown of the calculations in lines 780 and 850. These examples assume a value for L0 (positions 11 to 15 of file FLS) of 3200 or 670.

Example 1		*Example 2*	
L0=3200		L0=670	
3200		670	
INT(L0/66)	Integer portion of	INT(L0/66)	This determines the number
48	L0 divided by 66.	10	of pages already printed.
(48)*66	Integer portion of	(11)*66	This determines total lines
3168	L0 multiplied by 66.	660	already printed on pages.
(3168)+66	Lines printed on	(660)+66	This allows for inclusion of
3234	other pages plus 66.	726	the 66 lines available on the current page being printed.
(3234)-L0	Total lines	(726)-L0	This determines the line spaces
34	possible (including current page) minus L0.	56	remaining on the current page (34 and 56, respectively).

User Program Control

Printing can also be controlled by the user keeping track of the lines printed on each page.

```
0010 T=7                                  T = lines per page
0020 H=2                                  H = lines in the page heading
0030 S=T-H                                S = lines available for printing
0040 GOSUB 120
0050 FOR I=1 TO 110
0060 PRINT FLP,I <------------------------ Print your report
0070 L=L+1
0080 IF L=S GOTO 100 <-------------------- Test for printed lines equal to
0090 GOTO 110                              S = lines available for printing
0100 GOSUB 120
0110 NEXT I <------------------------------ Skip to a new page
0120 REM                                  { Print page heading
0130 P=P+1              0120-0170         { Set page number
0140 L=0                                  { Lines printed = 0
0150 PRINT FLP
0160 PRINT FLP,TAB(15),'PAGE      ';P
0170 RETURN
```

```
                    PAGE        1
1
2
3
4
5

                    PAGE        2
6
7
8
9
10

                    PAGE        3
11
12
13
14
15
```

For simplicity of illustration, a page size of 7 lines was used. This would normally be 66 for standard printed reports. Checks, invoices, and other documents would require different page sizes. The page heading would also be more extensive; however, the concept is the same.

By changing the value in variable T, you can quickly accommodate various sizes of paper for the same report.

LOCATING A CHARACTER IN A STRING

Another form of the computed GOTO statement uses the IDX intrinsic function, which allows you to determine the exact position of a specific character within a character string. For example, assuming that the operator entered N in response to input statement 920:

```
0910 DIM A$1
0920 PRINT 'ARE DIVIDENDS TO BE REINVESTED? Y OR N'
0930 INPUT A$
0940 GOTO 950,2000 ON IDX ('YN',A$)
0950 GOTO 910
0960 REM
  *
  *
  *
```

Statement 940 causes the program to branch to statement 2000 (the second statement number in the list, just as N is the second character in the string). If neither a Y or N is entered, the program repeats the prompt to the operator (statement 950).

TESTING FOR AN ERROR

The ONERROR statement provides another means of error recovery. This statement operates with two internal functions (&ERR and &LINE) to identify any type of error by error number and by the number of the line at which the error occurred. You can enter an ONERROR statement with a GOTO parameter to transfer program control to a particular statement in the event of an error, as shown below:

```
0010 ONERROR GOTO 115
0020 OPEN FL9,'D80',5,'GEORGE',OUT
0030 PRINT 'ENTER PRINCIPLE'
0040 INPUT P
0050 PRINT 'TIME','RATE','AMOUNT'
0060 FOR T=1 TO 10
0070 FOR R=1 TO 20
0080 A=P*(1+R/100)↑T
0090 PUT FL9,T,R,A
0100 NEXT R
0110 NEXT T
0115 ONERROR SYSTEM
0120 CLOSE FL9
0130 STOP
```

Calculated future value of principal at a rate of 1% to 20% for 1 to 10 years compounded yearly

In this example, the ONERROR statement specifies that file FL9 be closed (statement 120) if an error occurs. If, for example, the file was too small to hold all the values being entered into it, the ONERROR statement would ensure that the file was properly closed. The file could then be re-marked to a larger size.

The internal constants (&ERR and &LINE) can also be used to record error occurrences in a program with many input/output statements. In the following example, the ONERROR statement specifies that the program branch to the PRINT statement (line 150). The internal constants &ERR and &LINE are then inserted into the displayed line to indicate the error number and line number at which the error occurred.

```
0015 ONERROR GOTO 146
0020 OPEN FL4,'D80','AF',IN
0030 GET FL4,A,B,C,D,E
0040 LET B=A
0050 LET A=36
0060 LET C=C+B
0070 CLOSE FL4
0080 OPEN FL4,'D80','AF',OUT
0090 PUT FL4,A,B,C
0100 CLOSE FL4
0110 OPEN FL4,'D80','AF',IN
0120 GET FL4,A,B,C,D,E
0130 LET A=B*C
0140 LET D=A-E
0145 GOTO 160
0146 ONERROR SYSTEM
0150 PRINT 'ERROR',&ERR,'HAS OCCURRED AT LINE',&LINE
0160 CLOSE FL4
0170 STOP
```

Note that the EXIT statement and the error exit clauses on input/output statements take precedence over the ONERROR statement. In other words, if an EOF condition occurs in a statement with an EOF exit specified, the EOF exit is taken even though the program might also contain an ONERROR statement. ONERROR SYSTEM should be the first statement of an error recovery program to avoid loops. Terminal errors clear internal error pointers and the program must go to end-of-job.

SORTING AN INDEX FILE

When you create a key-indexed file, the key and corresponding location of each record in the file is stored in the index file. Subsequent access of the file can be significantly improved if you sort these keys into sequence. The following sample program illustrates a method of sorting the record keys in the index file. This is a storage sort and assumes that all keys can be loaded into storage at one time. The size of your machine will determine the maximum number of keys that can be sorted in this manner. If your 5110 storage size is less than 64K, adjust the DIM statements (130, 160, and 190) accordingly. Only those statements in the sample program that pertain to the index sort are discussed; others may be self-explanatory.

```
0010 REM INDEX FILE SORT PROGRAM
0020 REM
0030 REM
0040 REM IF STORAGE SIZE IS LESS THAN 64K ADJUST DIMENSIONS
0050 REM FOR K$ AND X ACCORDINGLY
0060 REM
0070 PRINT 'ENTER DEVICE CODE, FILE NUMBER AND FILE IDENT'
0080 PRINT 'FOR INDEX FILE TO BE SORTED'
0090 INPUT D$,F,F$
0100 OPEN FILE FL1,D$,F,F$,IN
0110 R=RLN('FL1')
0120 IF R≠32 GOTO 150
0130 DIM K$32(1400),X(1400)
0140 GOTO 200
0150 IF R≠16 GOTO 180
0160 DIM K$16(2300),X(2300)
0170 GOTO 200
0180 IF R≠8 GOTO 350
0190 DIM K$8(3500),X(3500)
0200 REM READ IN ALL RECORDS IN THE FILE
0210 I=I+1
0220 READFILE FL1,K$(I),EOF 240
0230 GOTO 210
0240 REM CLOSE INPUT FILE, AND DETERMINE SORTED ORDER
0250 I=I-1
0260 CLOSE FILE FL1
0270 MAT K$(I)=K$
0280 MAT X(I)=AIDX(K$)
0290 REM REWRITE INDEX FILE IN ASCENDING SEQUENCE
0300 OPEN FILE FL1,D$,F,F$,OUT,RECL=R
0310 FOR J=1 TO I
0320 WRITEFILE FL1,K$(X(J))
0330 NEXT J
0340 STOP
0350 PRINT 'NOT VALID INDEX FILE RECORD LENGTH'
```

Identify key index file.

Length of last record in FL1.

Key records are always 8, 16, or 32 bytes.

Space for 3500 8-byte keys in K$.

Bring all key records into storage.

Set matrix size to I elements.

Ascending index value of K$ into X.

New file of keys in K$ created as indexed by X(J).

Statements 70 and 80 request the indicated information, which is then used to open the file in statement 100. The length of the last record accessed in the file referenced FL1 is assigned to R in statement 110. In statements 120, 150, and 180, the exact record length is tested, and variables K$ and X are adjusted accordingly in the following statement. Note that, if record length is not equal to 8 (statement 180), the program terminates by branching to statement 350. Statements 210, 220, and 230 read all the record keys from the index file and branch to statement 240 when end of file is reached. The file is closed in statement 260. Statements 270 and 280 put the indexed file values in ascending order into matrix X (see *Index Function* in Chapter 2). Statement 300 reopens the original file (referenced FL1) for output with the same record length. Finally, statements 310, 320, and 330 write all the record key values into the file in ascending sequence according to their ascending order in matrix X.

ANOTHER WAY TO READ A STREAM INPUT FILE

Using a system file called file FLS, you can obtain an alternate form of stream-oriented file input. This form of input allows your program to get a logical record from a stream-oriented file and assign the entire record (including all quotation marks and commas) to one character variable. Thus, a BASIC program in source form and in a type 2 or type 9 file can be read and processed as in the preceding cross-reference program. Alternate stream file input can be obtained only from a file that is already open. You can invoke alternate file input by writing the file reference code (FL0-FL9) of the file in positions 7 through 9 of file FLS using the WRITE FILE statement (see the *IBM 5110 BASIC Reference Manual*).

A logical data statement in a stream I/O file can be read into a single variable as illustrated:

```
0010 :94560,'ADAMS Supply',964.60,359.00,'8/22/77'
```

The above record could represent:

Customer Number	94560
Customer Name	Adams Supply
Total Purchases to Date	964.60
Last Purchase	Amount 359.00
Date of Last Purchase	8/22/77

The data, if located in file 3 on device 'D80', could be read into a single variable as follows:

```
0010 OPEN FL1,'D80',3,'CUSTOMER',IN
0020 DIM A$64
0030 WRITEFILE USING 40,FLS,'FL1'
0040 FORM POS7,C
0050 GET FL1,A$
0060 PRINT A$
```

The data in A$ as printed in statement 60 would include all commas and quotes as well as the actual data. The output from 60 would appear:

```
94560,'ADAMS SUPPLY',964.60,359.00,'8/22/77'
```

Referring to the cross reference program earlier in this chapter, the program uses a type 2 or type 9 file for input as follows:

```
0080 ONERROR GOTO 200
0090 OPEN FILE FL1,D$,F,F$,IN
0100 ONERROR SYSTEM
0200 IF &ERR≠608 GOTO 270
0210 ONERROR SYSTEM
0220 OPEN FL1,D$,F,F$,IN
0230 WRITEFILE FLS,'          FL1'
```

If an error occurs, control transfers to statement 0200.

If this statement executes without error, a type 9 file has been opened.

Reset error trapping.

Error 608 indicates that a type 2 file exists and that the program tried to open it as a type 9 file. Control follows in statement 220, where the type 2 file is opened. Any other error passes control to statement 270 for termination.

Reset error trapping.

Open the type 2 file.

Invokes alternate input from a type 2 file to supply a logical record, including quotes and commas, in a single character varialbe.

With these statements, you have opened file FL1 and you have dimensioned A$ and M$ to contain the logical records. By testing for an error in the opening of a stream file as a record file, the program reads either a type 9 file (fixed length record) or a type 2 file (variable length record) with delimiter characters. This alternate mode ignores the commas of the type 2 file.

Different File Access Methods

The following programs were written to illustrate various methods of file accessing. Before these programs were written, the diskette was first marked. Unlike tape files, diskette files can be marked and then re-marked, if necessary, without affecting any other files.

Programs were created to illustrate the following record I/O topics:

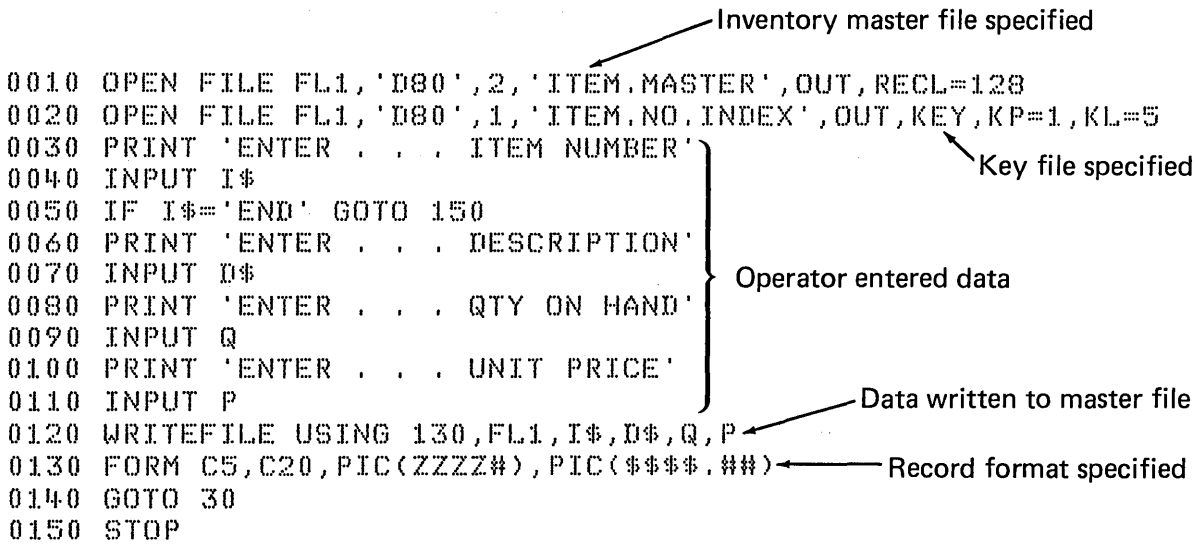
- Data entry with key index file
- Direct access and update with key index
- Sequential access by key index
- Sequential access with no key
- Direct access by relative record number
- Create multiple index

Each program uses the data file created by the first program.

Creating an Index File

The purpose of the following program is to illustrate the method by which a record I/O file with a key index file is created. In this example an inventory master file is created with a key file of item numbers. The amount of data is purposely limited, and the method of data entry is simplified in order to focus on the method by which both the master and key file are created.

Consider that you wish to store inventory data in your 5110. Fast access to each item is critical and the inventory data must be easily and quickly updated. The initial data files are created with the following program:




```
0010 OPEN FILE FL1,'D80',2,'ITEM.MASTER',OUT,RECL=128
0020 OPEN FILE FL1,'D80',1,'ITEM.NO.INDEX',OUT,KEY,KP=1,KL=5
0030 PRINT 'ENTER . . . ITEM NUMBER'
0040 INPUT I$
0050 IF I$='END' GOTO 150
0060 PRINT 'ENTER . . . DESCRIPTION'
0070 INPUT D$
0080 PRINT 'ENTER . . . QTY ON HAND'
0090 INPUT Q
0100 PRINT 'ENTER . . . UNIT PRICE'
0110 INPUT P
0120 WRITEFILE USING 130,FL1,I$,D$,Q,P
0130 FORM C5,C20,PIC(ZZZZ#),PIC($$$$.##)
0140 GOTO 30
0150 STOP
```

In the above example, the record format is specified in statement 130.

Item Number	5 characters
Description	20 characters
Quantity on Hand	5 characters
Unit Price	8 characters

The key field is automatically created as a result of statement 20, which specifies file 1 on device D80 as the key file using 5 characters (KL=5) starting at position 1 (KP=1) of the master file (item number) as the key. The overall record length is 128 bytes (RECL=128). Because our data consumes only 38 bytes, ample space is available for additional inventory data.

After the inventory data was entered, the master data file was listed and is shown below:

```
B202 SCREWDRIVER              5000    $1.25
A500 SCREWDRIVER                55    $8.50
B300 BOLTS                     600    $0.85
5000 PUMP 3/4 HORSE              7   $95.00
5002 PUMP 1/2 HORSE             12   $85.00
A202 HAMMER                     22    $6.25
A305 PIPE WRENCH                13   $16.75
```

The order in which the items are listed is the order in which they were originally entered. As you look at each record you can see the record format (space for 5-character item number, space for 20-character description, and so on.

This data file is now available for access and processing.

The key file created in file 1 should be sorted to produce the best operating performance.

Direct Access and Update with Key Index

The purpose of the following program is to illustrate how you can directly access a data file by key, alter the data, and update the master file. This program uses the data file created by the previous program. The method of altering the data after it has been accessed uses a simplified version of the full screen formatting capability of the 5110. Using these concepts, you should be able to construct efficient routines.

```
0010 Z=257
0020 W=Z+331
0030 DIM F$64,S$80
0040 F$='DESCRIPTION        ON HAND        UNIT PRICE'
0050 E$='ITEM NUMBER'
0060 STR(S$,1,43)='FORM POSZ,C11,X3,C5,X184,C47,X80,C1,C20,C2,'
0070 STR(S$,44,35)='PIC(ZZZZ#),C6,PIC($$$$$#.##),C1,POSW'
0080 REM
0090 OPEN FILE FL1,'D80',2,'ITEM.MASTER',ALL
0100 OPEN FILE FL1,'D80',1,'ITEM.NO.INDEX',ALL,KEY
0110 OPEN FILE FL2,'002',ALL
0120 PRINT 'ENTER INDEX KEY'
0130 INPUT K$
0140 IF K$='END' GOTO 230
0150 READFILE USING 160,FL1,KEY=K$,I$,D$,Q,P,NOKEY 250
0160 FORM C5,C20,NC5,NC8.2
0170 WRITEFILE USING S$,FL2,E$,I$,F$,'>',D$,'<>',Q,'<<<>>>',P,'<'
0180 READFILE USING 190,FL2,D$,Q,P,CONV 170
0190 FORM POSW,C20,X2,NC5,X6,NC8.2
0200 REWRITEFILE USING 210,FL1,D$,Q,P
0210 FORM POS6,C20,PIC(ZZZZ#),PIC($$$$$#.##)
0220 GOTO 120
0230 PRINT 'END OF JOB'
0240 STOP
0250 PRINT '************ NO KEY FOUND ************'
0260 PRINT
0270 GOTO 120
```

Constant for controlling cursor position

Specify format of inventory data display

Position cursor as specified in statement 10

Open master and key file for input and output

Open screen for record I/O input and output

Enter item number desired

Item is retrieved using key

Item data is displayed

Data items can be updated

Master file is updated

The master file is available for both access and update because the parameter ALL is specified in both OPEN statements 90 and 100. OPEN statement 110 referencing device 002 prepared the display screen for record input and output.

The operator keys in the item number in statement 130. The 5110 then searches the key file for that item number. When found, the key record points directly to the location of the data record in the inventory master file. The master file data is retrieved (statement 150) and displayed for operator viewing (statement 170). The system positions the cursor at the beginning of the description data field. If no changes are to be made, the operator simply presses the EXECUTE key. If changes are needed, the operator positions the cursor appropriately, keys the altered data, and presses the EXECUTE key. When the execute key is pressed, the displayed data is rewritten back to the master file.

```
ITEM NUMBER    5002

             DESCRIPTION                    ON HAND    UNIT PRICE

            >PUMP 1/2 HP              <>    13<<<>>>   $85.00<


5002                                                    0180    588

ITEM NUMBER    A202

             DESCRIPTION                    ON HAND    UNIT PRICE

            >HAMMER                   <>    18<<<>>>    $6.25<


A202                                                    0180    596
```

The above illustration is a copy of the display screen output. Item number 5002 was requested, and the quantity on hand has been altered to 13. Then item A202 was requested, and the quantity on hand was altered to 18.

If an invalid key item number was entered, the NOKEY parameter in statement 150 would cause the system to print 'NO KEY FOUND' (statement 250) and request the next key (statement 120). The marks to the left and right of each data field bracket the area where valid data may be entered.

Sequential Access by Key Index

The purpose of the following program is to illustrate how the data in a master file can be accessed in sequential order specified by the key. Sequential access by key means that each key and its corresponding record will be accessed in alphameric order. That is, all keys beginning with A are accessed first, then B, and so on with numeric-only keys accessed last.

```
0010 REM SAMPLE PROGRAM TO READ THE INVENTORY
0020 REM DATA BASE SEQUENTIALLY BY KEY ITEM NUMBER
0030 REM
0040 REM
0050 REM
0060 DIM F$64,H$64,D$20
0070 REM
0080 DATA 'ITEM NUMBER     DESCRIPTION     ON HAND      UNIT PRICE'
0090 READ H$
0100 PRINT FLP,H$
0110 REM
0120 STR(F$,1,39)='FORMPOS1,X3,C5,X7,C15,X1,PIC(ZZZZ#),X3,'
0130 STR(F$,40,18)='PIC($$$$$#.##),SKIP'
0140 REM
0150 OPEN FILE FL1,'D80',5,'ITEM.MASTER',IN
0160 OPEN FILE FL1,'D80',4,'ITEM.NO.INDEX',IN,KEY
0170 PRINT FLP
0180 READFILE USING 190,FL1,I$,D$,Q,P,EOF 220
0190 FORM C5,C20,NC5,NC8.2
0200 PRINT USING F$,FLP,I$,D$,Q,P
0210 GOTO 180
0220 PRINT TAB(30),'END OF JOB'
0230 STOP
```

Format output data.

Key and data files specified.

Data file is read until the last item is encountered

Each item is printed

ITEM NUMBER	DESCRIPTION	ON HAND	UNIT PRICE
A202	HAMMER	18	$6.25
A305	PIPE WRENCH	13	$16.75
A500	SCREWDRIVER	55	$8.50
B202	SCREWDRIVER	5000	$1.25
B300	BOLTS	600	$0.85
5000	PUMP 3/4 HORSE	7	$95.00
5002	PUMP 1/2 HORSE	13	$85.00

Running the program shown above automatically creates the printed output as illustrated. Notice that the item numbers are listed alphamerically, and that the on-hand quantities have been updated according to the previous program.

If the key file was sorted, the next sequential key could be located more quickly, thus making this program execute much faster.

Sequential Access with No Key Used

The purpose of this program is to illustrate how your data file may be accessed sequentially without the use of a key index file. Even though a key file was created, it is not necessary to use it for every access. Sequential access to a master file without a key simply means to retrieve the records one after the other in their order of appearance in the file. In many cases, this will be their original order of entry.

The following program is the same as the previous program except that statement 160 has been deleted; you do not open the key file. The result is a listing of items exactly as they appear in the master file.

```
'0010 REM SAMPLE PROGRAM TO READ THE INVENTORY
0020 REM DATA BASE SEQUENTIALLY . . NO KEY FILE
0030 REM
0040 REM
0050 REM
0060 DIM F$64,H$64,D$20
0070 REM
0080 DATA 'ITEM NUMBER   DESCRIPTION      ON HAND    UNIT PRICE'
0090 READ H$
0100 PRINT FLP,H$
0110 REM
0120 STR(F$,1,39)='FORMPOS1,X3,C5,X6,C12,X5,PIC(ZZZZ#),X5,'
0130 STR(F$,40,18)='PIC($$$$$.##),SKIP'
0140 REM
0150 OPEN FILE FL1,'D80',2,'ITEM.MASTER',IN
0170 PRINT FLP
0180 READFILE USING 190,FL1,I$,D$,Q,P,EOF 220
0190 FORM C5,C20,NC5,NC8.2
0200 PRINT USING F$,FLP,I$,D$,Q,P
0210 GOTO 180
0220 PRINT TAB(30),'END OF JOB'
0230 STOP
```

The output of the above program is illustrated below:

```
 ITEM NUMBER   DESCRIPTION      ON HAND    UNIT PRICE

    B202       NUTS              5000        $1.25
    A500       SCREWDRIVER         55        $8.50
    B300       BOLTS              600        $0.85
    5000       PUMP 1/2 HP         13       $95.00
    5002       PUMP 3/4 HP          7       $85.00
    A202       HAMMER              18        $6.25
    A305       PIPE WRENCH         13       $16.75
```

This technique is handy for creating a fast listing of a data file because it avoids access to the key file.

Direct Access by Relative Record Number

The purpose of the following program is to illustrate how you can access a record directly if you know its location in the master file. This is the fastest method of access because the system can go directly to the desired record in the master file rather than looking up the location in a key file or searching for the record sequentially.

The following program is the same as the program previously described under *Random Access and Update Using Key Index* with the following changes:

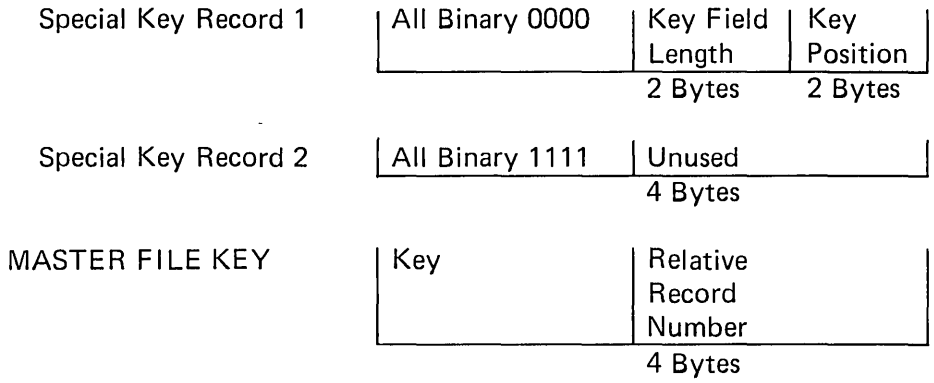
Statement 100 deleted	No key file is opened and specified.
Statements 120 & 130	The operator enters a numeric record number rather than a key.
Statement 150	The record number is specified in the read statement with a REC= clause and a NOREC error branch.
Statement 250	Error NO RECORD FOUND is displayed.
Statement 140	Branch on zero rather than END.

The program and its output are shown below: Notice that record number 3 is actually the third record entered by the original data entry program.

```
0010 Z=257
0020 W=Z+331
0030 DIM F$64,S$80
0040 F$='DESCRIPTION                ON HAND    UNIT PRICE'
0050 E$='ITEM NUMBER'
0060 STR(S$,1,43)='FORM POSZ,C11,X3,C5,X184,C47,X80,C1,C20,C2,'
0070 STR(S$,44,35)='PIC(ZZZZ#),C6,PIC($$$$$.##),C1,POSW'
0080 REM
0090 OPEN FILE FL1,'D80',2,'ITEM.MASTER',ALL
0110 OPEN FILE FL2,'002',ALL
0120 PRINT 'ENTER RECORD NUMBER'
0130 INPUT K
0140 IF K=0 GOTO 230
0150 READFILE USING 160,FL1,REC=K,I$,D$,Q,P,NOREC 250
0160 FORM C5,C20,NC5,NC8.2
0170 WRITEFILE USING S$,FL2,E$,I$,F$,'>',D$,'<>',Q,'<<<>>>',P,'<'
0180 READFILE USING 190,FL2,D$,Q,P,CONV 170
0190 FORM POSW,C20,X2,NC5,X6,NC8.2
0200 REWRITEFILE USING 210,FL1,D$,Q,P
0210 FORM POS6,C20,PIC(ZZZZ#),PIC($$$$$.##)
0220 GOTO 120
0230 PRINT 'END OF JOB'
0240 STOP
0250 PRINT '************ NO RECORD FOUND ************'
0260 PRINT
0270 GOTO 120

ITEM NUMBER    B300

            DESCRIPTION                ON HAND    UNIT PRICE

           >BOLTS                   <>   600<<<>>>    $0.85<



3
ENTER RECORD NUMBER

?
                                                        0130    001
```

Create Multiple Index

It may be desirable to have several key index files for a single master file. The 5110 can create one index file automatically as illustrated in the first data entry program. Suppose you wish to create a report organized alphabetically by item description. A second key file can be created with the item description as the key field using the following program. Two special key records (marker records) are required in the first two record locations of an unsorted key file (see the *IBM 5110 BASIC Reference Manual, Index file format*). This program builds the first two special key records and all subsequent keys for the master file.



```
0010 REM CREATE INDEX FILE FOR EXISTING MASTER
0020 REM
0030 DIM K$28,L$28,D$3,F$17
0040 PRINT 'ENTER DEVICE CODE, FILE NUMBER AND FILE NAME'
0050 PRINT 'FOR THE MASTER FILE TO BE USED.'
0060 INPUT D$,F,F$
0070 OPEN FILE FL1,D$,F,F$,IN
0080 R=RLN('FL1')
0090 REM
0100 REM GET KEY INFORMATION
0110 PRINT 'ENTER KEY LENGTH AND KEY POSITION.
0120 INPUT L,P
0130 REM
0140 REM CHECK FOR VALIDITY
0150 IF L≠INT(L)!P≠INT(P) GOTO 590
0160 IF L<1!P<1 GOTO 590
0170 IF R<P+L-1 GOTO 590
0180 REM
0190 REM DETERMINE KEY RECORD SIZE
0200 R1=32
0210 IF L≥13 GOTO 250
0220 R1=16
0230 IF L≥5 GOTO 250
0240 R1=8
0250 REM
0260 REM GET INFORMATION FOR INDEX FILE
0270 REM
0280 PRINT 'ENTER DEVICE CODE, FILE NUMBER AND FILE NAME'
0290 PRINT 'FOR INDEX FILE TO BE BUILT'
0300 INPUT D$,F,F$
0310 OPEN FILE FL2,D$,F,F$,OUT,RECL=R1
0320 R2=R1-3
0330 S=0
0340 K$=X'00'
0350 STR(K$,2)=K$
0360 WRITEFILE USING 370,FL2,STR(K$,1,L),L,P
0370 FORM C,POSR2,B2,B2
0380 REM
0390 REM GET KEYS FROM MASTER
0400 REM
0410 L$=K$
0420 READFILE USING 430,FL1,STR(K$,1,L),EOF 540
0430 FORM POSP,C
0440 R3=&REC
0450 IF S=1!K$>L$ GOTO 500
0460 L$=X'FF'
0470 STR(L$,2)=L$
0480 WRITEFILE FL2,STR(L$,1,L)
0490 S=1
0500 REM
0510 WRITEFILE USING 520,FL2,STR(K$,1,L),R3
0520 FORM C,POSR2,B4
0530 GOTO 410
0540 IF S=1 GOTO 580
0550 L$=X'FF'
0560 STR(L$,2)=L$
0570 WRITEFILE FL2,STR(L$,1,L)
0580 STOP
0590 PRINT 'INVALID KEY LENGTH OR KEY POSITION OR NOT VALID FOR'
0600 PRINT 'MASTER FILE RECORD LENGTH'
```

alphabet extender: Any one of the following three special characters: #, @, and $.

alphabetic character: Any of the 26 letters (A through Z) of the English alphabet or any of the alphabet extenders (#, @, and $).

alphameric character: A numeric or alphabetic character.

argument: An arithmetic expression appearing in parentheses following a function name, either in a function reference (either a user-written or an intrinsic function) or in a pseudo variable. The expression represents a value that the function is to act upon. The function name may or may not be followed by arguments.

arithmetic array: A named table of arithmetic data items. An array may be implicitly declared through usage or explicitly declared in a DIM statement. BASIC allows one- and two-dimensional arithmetic arrays.

arithmetic constant: A constant with a numeric value. The three forms of arithmetic constants permitted in BASIC are integer, fixed-point, and floating-point.

arithmetic data item: Data having a numeric value.

arithmetic expression: An arithmetic constant, a simple arithmetic variable, a scalar reference to an arithmetic array, an arithmetic-valued function reference, or a sequence of the above appropriately separated by arithmetic operators and parentheses.

arithmetic operator: A symbol representing an operation to be performed upon arithmetic data. The arithmetic operators are:

+	Addition and unary plus sign
-	Subtraction and unary minus sign
*	Multiplication
/	Division
↑ or **	Exponentiation

arithmetic variable: The name of an arithmetic data item whose value is assigned and/or changed during program execution. The name consists of a single alphabetic character or an alphabetic character followed by a digit.

array: A named list or table of data items, all of which are the same type–arithmetic or character. BASIC allows one- and two-dimensional arrays.

array declaration: The process of naming an array and assigning dimensions to it either explicitly (by the DIM statement) or implicitly through usage.

array element: See *array member*.

array expression: An arithmetic expression or a character expression representing an array of values rather than a single value. It may be used only in an array assignment statement.

array member: A single data item in an array; its position is indicated by a subscripted array reference.

array variable: The name of an entire array. The name consists of an alphabetic character (for arithmetic arrays) or an alphabetic character followed by the dollar sign, $, (for character arrays).

assignment: The process of giving values to variables; for example, via LET statements, READ statements, and INPUT statements.

assignment symbol: The symbol =, which is used in an assignment statement to give a value to one or more variables.

BASIC: A programming language designed for interactive systems and originally developed at Dartmouth College to encourage nonprogrammers to use computers for simple problem-solving operations. The word BASIC is an acronym for Beginners' All-purpose Symbolic Instruction Code.

binary operator: A symbol representing an operation to be performed upon two data items, arrays, or expressions. The four types of binary operators are arithmetic, character, logical, and relational.

branching: Executing a statement other than the next sequential one; for example, via the GOTO statement.

built-in function: See *intrinsic function.*

character array: A named table of character data items. An array may be implicitly declared through usage or explicitly declared in a DIM statement. BASIC allows one- and two-dimensional character arrays.

character constant: A constant with a character value. It is always enclosed by a pair of single or double quotation marks.

character data: Data having a character value as opposed to a numeric value.

character expression: A character constant, a simple character variable, a scalar reference to a character array, a character-valued function reference, or a sequence of the above separated by the concatenation operator (| |) and parentheses.

character operator: A symbol representing an operation to be performed upon character data. The concatenation operator (| |) is the only character operator in BASIC.

character string: A sequence of characters that represents an item of character data.

character variable: The name of a character data item whose value is assigned and/or changed during program execution. The name consists of an alphabetic character followed by the dollar sign character ($).

comment: A remark or note included in the body of a program by the programmer. It has *no* effect on the execution of the program; it merely documents the program. Comments are written as a string of characters and may appear as a part of any program statement that has no operands (for example, REM, STOP, END, and RESTORE).

concatenation: The joining of two character data items by the symbol | | .

concatenation operator: The symbol | |, used to concatenate, or join, two character data items.

constant: A value that never changes. BASIC has two types of constants: arithmetic and character.

control specification: (1) One of the specifications X or POS, used in the FORM statement to specify formatting of records in record-oriented files. (2) One of the specifications X, POS, or SKIP, used in the FORM statement to control print line formatting.

data file: See *file.*

data form specification: (1) One of the specifications B, C, NC, PD, S, L, or PIC, used in the FORM statement to specify formatting of character and arithmetic values in record-oriented files. (2) One of the specifications C or PIC, used in the FORM statement to format character and arithmetic values on a printed line.

data item: A single unit of data; that is, a constant, a variable, an array element, or a function reference.

data table: The values contained in the DATA statements of your program. DATA statements are processed in statement number sequence (lowest to highest). The values in each DATA statement are collected and placed in a single table in order of their appearance (left to right).

data table pointer: An indicator that moves sequentially through the data table, pointing to each value as it is assigned to a corresponding variable in a READ statement. Initially, the indicator refers to the first item in the table. It can be repositioned to the beginning of the table at any time by the RESTORE statement.

declaration: See *explicit declaration* and *implicit declaration*.

delimiter: A character that groups or separates data items.

digits: the numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

dimension specification: The specification of the size of an array and the arrangement of its members into one or two dimensions.

direct access: The storage or retrieval of data independently of other data in a file (that is, regardless of its location relative to other data).

dummy variable: A simple variable enclosed in parentheses and placed after the name of a user-written function in a DEF statement. The function performs its defined calculation on the expression value substituted for each dummy variable when the program is executed.

E-format: Floating-point format.

EBCDIC collating sequence: The ordering of character data items according to the Extended Binary Coded Decimal Interchange Code.

error message: A message generated by the computer when an error has been detected.

executable statement: A program statement that causes an action to be performed by the computer.

execution error: An error discovered during execution of a BASIC program (for example, dividing by zero, or branching to a nonexisting statement number).

explicit declaration: The use of a DIM statement to specify the number of members in an array, the number of dimensions in an array, or the length of a character variable.

exponent (of E-format number): An integer constant specifying the power of ten by which the base (mantissa) of the decimal floating-point number is to be multiplied.

exponentiation: Raising a value to a power.

expression: A representation of a value; for example, variables and constants appearing alone or in combination with operators. Three forms of expressions are defined in BASIC: scalar (arithmetic or character), array (arithmetic or character), and logical.

extended alphabet: The 26 letters of the English alphabet and the 3 alphabet extenders ($ # @).

F-format: Fixed-point format.

file: A named group of related data items that are stored together. In BASIC there are two types of files: stream-oriented and record-oriented.

file reference: FL0 through FL9.

fixed-point constant: An arithmetic constant consisting of one or more digits and a decimal point, and optionally preceded by a sign.

fixed-point format: The form used to express a fixed-point constant.

floating-point constant: An arithmetic constant consisting of an integer or fixed-point constant followed by the letter E, followed by an optionally signed one- or two-digit integer constant.

floating-point format: The form used to express a floating-point constant.

full print zone: Eighteen horizontal print positions. In a PRINT statement, a comma is used to indicate that a full print zone should be used.

function: A named expression that computes a single value. See also *intrinsic function* and *user-written function.*

function reference: The appearance of an intrinsic function name or a user-written function name in an expression.

generic key: An argument specified in the KEY clause of a record I/O statement that is less than the full key length defined for a corresponding file.

I-format: Integer format.

implicit declaration: (1) The specification of the number of members in an array or the number of dimensions in an array, either by a reference to a member of an array or by context (without the array being explicitly specified in a DIM statement). (2) The specification of the length of a character variable by context (without the variable being explicitly defined in a DIM statement).

input: The transfer of data from an external medium to internal storage.

input list: A list of variables to which values are assigned from input data; the list can be made up of scalar variables, array member references, pseudo variables, array references, and array references with redimensioning.

input/output: The transfer of data between an external medium (that is, the keyboard or a file) and internal storage.

integer constant: An arithmetic constant containing one or more digits, optionally preceded by a sign.

integer format: The form used to express an integer constant.

internal constant: An arithmetic constant whose value is supplied by BASIC. The name of the internal constants are &PI, &SQR2, &E, &INCM, &LBKG, and &GALI.

internal storage: A computer's main storage.

intrinsic function: A function supplied by BASIC (for example, SIN, COS, or SQR).

key: One or more consecutive characters used to identify a particular record in a key-sequenced file.

key-sequenced file: A record-oriented file whose records are accessed according to keys.

logical expression: A logical subexpression, or two logical subexpressions joined by a logical operator (& or |). Its value is either true or false.

logical operator: An operator that is used in a logical expression. The logical operators are: & (AND) and | (OR).

long-form precision: Precision whereby, externally, values printed with I-format and F-format have a maximum of 15 significant digits, and values printed with E-format have a maximum of 15 significant digits in the mantissa.

loop: A sequence of instructions that is executed repeatedly until a terminating condition is reached. The FOR statement identifies the beginning of a loop; the NEXT statement identifies the end of it.

mantissa: In floating-point notation (E-format), the number that precedes the E. The value represented is the product of the mantissa and that power of ten specified by the exponent.

matrix (mathematical): A two-dimensional arithmetic array.

multiline function: A user-defined function that is defined with more than one statement.

nesting: (1) The occurrence of a FOR/NEXT loop within another FOR/NEXT loop. (2) The occurrence of a GOSUB statement when one or more GOSUB statements are already active. (3) The use of more than one set of parentheses to indicate the order of evaluation in a complex arithmetic expression.

nonexecutable statement: A program statement that specifies information for program execution.

null character string: Two adjacent single quotation marks that specify a character constant of blank characters.

null delimiter: One or more blanks or no characters at all (that is, one data item directly following another data item with no intervening space or delimiter) used in a PRINT statement to specify a packed print zone.

numeric character: Any of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

operand: A constant, a variable, an array member reference, a function reference, or a subexpression on which an operation is to be performed.

operator: A symbol specifying an operation to be performed. See also *arithmetic operator, binary operator, concatenation operator, logical operator, relational operator,* and *unary operator.*

output: The transfer of data from internal storage to an external medium.

output list: A list of variables from which values are written to an output file; the list can be made up of scalar expressions and array references.

packed print zone: A section of a printed line, consisting of a number of horizontal print positions, whose size is determined by the type (arithmetic or character) and length of the data being printed. In the PRINT statement, a semicolon or null delimiter is used to indicate that a packed print zone is to be used.

padding: The addition of one or more blanks to the right of a character string to extend the string to a required length.

precision: The number of digits for which significance can be expressed.

print zone: See *full print zone* and *packed print zone.*

priority: A rank assigned to an arithmetic operator; it is used when an arithmetic expression is being evaluated. The order of priorities, from high to low, is exponentiation, unary operations, multiplication and division, addition and subtraction. Operations at the same priority level are evaluated as they are encountered (from left to right in the expression).

program: A logically self-contained sequence of BASIC statements that can be executed by the computer to attain a specific result.

programmer-defined function: See *user-written function.*

pseudo variable: The use of an intrinsic function as a receiving variable. STR is the only pseudo variable in BASIC.

record: A collection of related data items treated as a unit.

record-oriented file: A file in which items are stored in records.

redimension specification: The assignment of a new dimension specification to an already existing array, via an array assignment statement, a READ statement, an INPUT statement, a GET statement, a READ FILE statement, or a REREAD FILE statement.

redimensioning: The changing of the number of dimensions or the number of members in each dimension of a previously declared array.

relational operator: An operator used in a logical subexpression. The relational operators are:

=	Equal to
≠ or < >	Not equal to
>	Greater than
<	Less than
> = or ≥	Greater than or equal to
< = or ≤	Less than or equal to

remark: See *comment*.

scalar: A single data item (as opposed to an array of items).

scalar expression: An arithmetic expression or a character expression representing a single value rather than an array of values.

sequential access: The retrieval of data according to the order in which the data is stored in a file.

short-form precision: Precision whereby, externally, values printed with I-format and F-format have a maximum of seven significant digits, and values printed with E-format have a maximum of seven significant digits in the mantissa.

significant digits: All the digits of a number starting with the leftmost nonzero digit.

simple name: Any combination of up to 8 alphabetic and numeric characters (with no blanks).

simple variable: A scalar variable (but not an array member).

single-line function: A user-defined function that is defined in one statement (that is, the DEF statement).

special characters: Any characters allowed in BASIC that are not alphameric characters.

statement number: The number that prefaces a BASIC statement. It can be up to four digits in length (in the range 0000 to 9999).

stream-oriented file: A file in which items are stored as a stream of data and retrieved in sequential order.

subexpression: A group within an arithmetic expression and used by the computer to evaluate that expression.

subroutine: A program segment (sequence of statements) branched to by a GOSUB statement. The last statement of a subroutine must be a RETURN statement that directs the computer to return and execute the statement following the GOSUB statement.

subscript: Any valid arithmetic expression (whose truncated integer value is greater than zero) used to refer to a particular member of an array.

substring: A part of a character string.

system-supplied constants: See *internal constants*.

truncation: The deletion of one or more characters on the right of a character string to shorten the string to a required length.

unary operator: An operator that precedes, and thus is associated with, an arithmetic expression. The unary operators are + (positive) and - (negative).

user: Anyone utilizing the services of a computing system.

user-written function: A function defined by the user in a single-line or multiline function definition.

variable: A name used to represent a data item whose value may change during execution of a program.

zero suppression: The elimination of leading nonsignificant zeros in a number.

IBM®

Technical Newsletter

This Newsletter No. SN21-0277

Date 28 April 1978

Base Publication No. SA21-9307-0

File No. None

Previous Newsletters None

filed
6/18/78

IBM 5110
BASIC User's Guide

This technical newsletter provides replacement pages for the subject publication. Pages to be inserted and/or removed are:

33 through 36	119, 120
41, 42	129, 130
51, 52	155 through 162
65, 66	165 through 168
77 through 80	175, 176
101, 102	181, 182

Changes to text and illustrations are indicated by a vertical line at the left of the change.

Summary of Amendments

Additions and corrections have been made to improve the accuracy and readability of the text.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901

Printed in U.S.A.

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.

SA21-9307-0

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 BASIC User's Guide Printed in U.S.A. SA21-9307-0

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 BASIC User's Guide Printed in U.S.A. SA21-9307

SA21-9307