```
+---------------------------------------+
|                                       |
|      B A S I C   T R A I N I N G       |
|                                       |
|         A GW-BASIC Tutorial           |
|          by Steve Estvanik            |
|   HTML Conversion by Thomas Antoni    |
|                                       |
+---------------------------------------+
```

Contents Table
==========

Preface
1. Introduction
    1.1 Basic History
    1.2 Getting Started
    1.3 Exercises
2. Structure and Style
    2.1 More BASIC Controls
    2.2 IF THEN Statements
    2.3 Check Book Balancer
    2.4 Summary
    2.5 Exercises
3. Loops, Colors and Sound
    3.1 Looping
    3.2 Exercise: Make this game more interactive
    3.3 Colors
    3.4 Exercise
    3.5 Sounds
4. Arrays and DATA Statements
    4.1 Arrays
    4.2 Exercises
    4.3 DATA Statements
    4.4 Exercises
    4.5 Final Exercise
5. GOSUBS and FUNCTIONS
    5.1 Why GOSUBS?
    5.2 ... A POLITICAL ANNOUNCEMENT
    5.3 Renumbering (RENUM)
    5.4 Exercise
    5.5 FUNCTIONS
    5.7 More Exercises
    5.8 Exercise using GOSUBS or FUNCTIONS
    5.9 Extra Credit
6. Files
    6.1 Sequential versus Random Files
    6.2 What's in a file?
    6.3 Using Files
    6.4 Sequential Details
    6.5 Notes on Random Files
    6.6 Questions and Answers
    6.7 Projects
7. Simple Graphics
    7.1 WIDTH
    7.2 SCREEN
    7.3 Exercise
    7.4 More Exercises
    7.5 For Super-Extra Credit
    7.6 The End of the Beginning

Preface
==========

This training course shows you the elements of the Basic programming language. Basic Training covers the essentials to get you started. Advanced Basic (available only to registered users), picks up from that point and covers areas such as animation, error trapping and real time event programming. (Run the REGISTER program for details on how to obtain the second part.)

The dialect of Basic that I use in this first part is based on the extended Basic named GW-BASIC or BASICA that comes with the IBM PC and most compatibles. The tutorial includes many examples. The source code is also included with each tutorial, so you can run it, modify it and experiment with it. I recommend that you make a backup copy or rename the source files, though, before starting to experiment. All the programs in this first part, and most of those in the second can be run using just an interpreter, but if

you're serious about learning Basic, I strongly recommend that you purchase one of the compilers.

[... back to the Contents Table]

1. Introduction

1.1 Basic History

Basic was invented at Dartmouth in the 50's as a response to the problems of instructing a computer. Previously, it was necessary for a programmer to understand the computer at the machine level before you could get it to do anything useful. This was much like requiring a course in auto mechanics before you could hire a taxi. With Basic, you gain an assistant who "interprets" their instructions and gets them where you want to go without worrying about the "fiddly bits". Basic is only one of a series of "languages" which have been developed for this purpose. In its original form, it was limited compared to richer languages such as Fortran or Pascal. In recent years, especially since the advent of microcomputers, Basic has been enhanced and can now stand as tall as any of the other languages. Each language has its advocates. In my daily consulting work and game design programming, I use many languages routinely. Basic remains one of my choices when I need to write a quick program of any sort, especially if it calls for graphics. This easy-to-write quality combined with its simple structure makes Basic a prime candidate for your first computer language. If you already speak one computer language, learning Basic is an easy way to expand your fluency.

[... back to the Contents Table]

1.2 Getting Started

Throughout this tutorial, I try to use commands that are common across all Basics. When that's not possible (for example with graphics and sound), I use Microsoft Basic for the IBM PC. Most other Basics used on IBM compatibles are identical, but occasionally there are small differences. Consult your user's manual to check on inconsistencies. When discussing compilers, I use the TurboBasic/PowerBasic syntax.

To get started, we'll create a simple, 2 line program. The program itself is less important than the mechanics of how to create, save and run a Basic program. Don't worry about what each line does right now, we'll cover that soon.

Start your version of Basic. If you're using an interpreter, just type:

```
   | > basica    |
or
   | > gw-basic  |
```

This loads interpreter Basic and you'll see the OK prompt. Depending on the version of DOS you have on your computer, there may be a different name for the basic interpreter. Check your DOS manual for the actual command name.

If you're running a compiler such as PowerBasic or QuickBasic, start the program, and run the examples in interactive mode (alt-R). (There's no OK prompt when you use these products.)

```
|                                                                        |
| As an incentive, we'll send you a copy of the Liberty Basic compiler for |
| Windows when you register.  This compiler lets you run Basic in the Windows |
| environment without the need to learn all the complexities of Windows |
| programming.  Print out the order form for details.  Enter the command: |
|      REGISTER   at the DOS prompt.                                     |
|                                                                        |
```

Now type the following 2 lines:

```
   | 10 PRINT "hello there...."  |
   | 20    GOTO 10               |
```

To test it out, type

```
   | RUN                         |
```

or, if using a compiler, press alt-R (or otherwise invoke the Run command).

You should see your message repeated endlessly. Since your computer has more patience than you for this sort of eternal game, press the <control><break> keys to end the program. You've just written and run a complete Basic program. This short program shows two quite useful statements. "Print" causes anything coming after it to be echoed to the screen. "Goto" tells the

program to jump to the line indicated (in this case 10). Thus our program will print the 2 words, then execute line 20. Since this line tells it to go back to line 10, we get what's termed an "infinite loop". Usually this is the result of a programming error or bug.

This short program also introduces the concept of line numbers. These numbers let the interpreter know the order in which to try to execute the commands. It also gives you a reference for GOTO commands. Interpreters require line numbers. Compilers can use them or not, as you like. Modern practise is to avoid them whenever possible, but I'll include them occasionally for backwards compatibility.

Our first program lacks elegance, and accomplishes nothing of much use, but it gives us something to build on. More importantly, it gives us something to save! You can save it now, by typing:

```
| SAVE "Hello"                 |
```

for the interpreter, or by using the File command, and the Save option if in a compiler.

Then leave the Basic interpreter by typing

```
|  SYSTEM  |
```

Leave the compiler by typing alt-X.

If you want to check that your program is really there, use the directory command:

```
| DIR H*.*                     |
```

You should see "Hello.bas" (plus any other programs you have that start with "h"). Note that Basic automatically assigns an extension of ".bas" to any program you save. You can override this, but good programming practise recommends against this. It will be easier to keep track of you rapidly increasing selection of programs if you keep the extensions meaningful. To run the program you have two choices. If you want to jump right into the program, you just type:

```
| BASICA Hello                 |
```

This will load the Basic program and immediately begin to run your program. Again, you'll need to break in order to stop this monster. Another approach would be to load Basic first, then issue the command:

```
| RUN "Hello"                  |
```

Note that this second case requires the use of quotes, but the result is the same, an endless stream of hello's.

The next element we'll look at is the assignment statement. This allows us to set the value of variables. Basic has three main categories of elements:

```
| Reserved words |-- special words like "print" and "goto"
     that have a special meaning to the interpreter

| Constants | -- numerical and string values like:
     2, 3.5 and "Hello there...".   (Note that string values
     use quote marks to show their beginning and end.)

| Variables | -- anything else.
```

That's not quite fair, but it's close. Variables are what give life to an otherwise rather boring collection of reserved words and constants (even their names are boring!). A variable is what you create to represent some other value. Think of them as containers that can hold many values. For example, if I use the variable X, I can put many values into it. This process is accomplished with the "LET" statement:

```
| LET X = 10  |
```

This says that X now has the value of 10. In fact very few programmers bother with the LET statement (Many programmers have yet to discover that you can use more than 2 fingers at a time on the keyboard, so they tend to be brief.) A terser, and more common way is just to say:

```
| X = 10      |
```

This may look familiar to you if you've taken algebra or new-math type courses. However, be careful! The "=" sign should be read as "is assigned"

or "takes the value of". Consider the following:

```
| X = X + 1   |
```

This says that X is now to be given the value of whatever X was before, plus 1. In this case, 11. One of the advantages of an interpreter is that it can give you an immediate response without running a program. You don't need special commands to use this mode, just don't use line numbers. Thus if you typed the initial line of our program, the interpreter would respond:

```
| Hello there...  |
```

Note that the quotes have disappeared. One use you can make of this feature is as a calculator. If you type any mathmatical expression, the interpreter will give you an answer. Eg, try the following to see how each is interpreted:

```
| 2+1       |
| 2*3       |
| 4/2       |
| 4-2       |
| 2+4*5     |
| (2+4)*5   |
```

Experiment until you feel comfortable with the way that arithmetic is handled by the interpreter. Notice, particularly, that the order of interpretation depends on both the operator and any parentheses. This is called precedence.

Basic also uses some defaults to describe whether a variable is a string, integer or real number. Strings are indicated by a '$' as final digit, integers as a '%', and real values as '!'. You tell Basic to set this default with the command:

```
| DEFINT A-Z  |
```

This says any variable starting with the letters A to Z is an integer, whether it has a '%' at the end or not. You could also define specific variables to be strings or integers, but that starts to get confusing, so I use the convention that all variables are integers unless otherwise indicated.

Now let's see how this works in a program. We'll start with our hello program, already in memory, and add to it:

```
| 10 PRINT "hello there....";X |
| 15 X = X + 1                 |
| 20    GOTO 10                |
```

Before trying this, make a prediction about what will happen. You should always try to predict what your program is going to do. Remembering that prediction will help when it does something you didn't anticipate.

[... back to the Contents Table]

1.3 Exercises

1. Experiment with various combinations of arithmetic operations, until you can predict what the results will be. In particular, try examples which mix multiplication and addition. What is the effect of using parentheses?

2. Use assignment statements to hold intermediate results. EG,

```
|  X = 2          |
|  Y = 3          |
|                 |
|  X + Y          |
|  X = X + Y      |
|  PRINT X, Y     |
```

(Note that when you assign a value, it isn't printed.)

[... back to the Contents Table]

2. Structure and Style

In the last chapter we looked at the most elementary statements in the Basic language. This time, we'll introduce the concept of branching and program control. In the next few installments we'll add loops, color, sound and arrays. At that point you'll be well equipped to start writing actual programs.

2.1 More BASIC Controls

A few more commands useful from within the interpreter:

```
| LOAD "myprog" |  --- gets your program from the disk
| LIST          |  --- lists the entire program
| LIST i - j    |  --- lists only lines i to j
| LLIST         |  --- lists the program to your printer
| RENUM         |  --- renumbers your program.  any relations within
                       your program are preserved.
```

The compiler doesn't need line numbers, and you can skip them entirely, so most of these commands aren't needed. Compilers have direct commands for printing a file (usually under the FILE menu item.). I'll continue to use line numbers in this tutorial, but my reason for doing so is that it's easier to discuss a program when I can refer to particular line numbers. For actual programs, I'll usually skip them.

Structured programming is just a small part of the overall concept of structured design. The root concept is to build a program from the top down. It also discards the concept of flow charts. Here's the reason: Flow charts are unnecessarily detailed. By the time you break a problem down to flow chart level, you can usually write the code directly. They're so detailed, that few programmers go back and change them when the program changes, so flow charts are usually out of date as soon as they're written. Structured programming does use diagramming techniques, but for our purposes we'll be able to use pseudocode for all our designs. We approach the problem by defining what we want to do at a very high level. For example, to create a checkbook balancer, we might outline the following steps:

```
|   Enter initial balance     |
|   Add deposits              |
|   Subtract checks           |
|   Show final balance        |
|   Quit                      |
```

Now we can elaborate this to describe each step in more detail:

```
|                                                       |
|   Enter initial balance                               |
|   Add deposits                                        |
|       -- ask user if there are any more deposits      |
|       -- if there are then                            |
|             get the next deposit                      |
|             add that amount to the balance            |
|       .... repeat as necessary                        |
|   Subtract checks                                     |
|       -- ask user if there are any more checks        |
|       -- if there are then                            |
|             get the next check                        |
|             subtract that amount from the balance     |
|       .... repeat as necessary                        |
|   Show final balance                                  |
|   Quit                                                |
|                                                       |
```

Note that some items require more elaboration than others. In some cases, we could write the Basic code directly, so there's no need for further refinement. Now, we're ready to write the actual code. Using this structured, top down approach any problems become apparent from the start. This text based way of describing a program is termed pseudocode, because it's simpler than English, but not rigorous enough to feed to the computer. It forms a useful link between human and computer. Well written pseudocode is easily converted to any language, and forms an outline of the program. It also suggests a preliminary set of comments.

[... back to the Contents Table]

2.2 IF THEN Statements

All programming statements that we'll look at are merely refinements or alternate forms of 3 basic forms: assignments, loops and conditional branching. Using only these 3 constructs, any possible program can be written. IF THEN, combined with the GOTO is an example of the third type of statement, the branch. In its simplest form,

```
| IF { conditional statement } THEN |
|    { statement 1 }                |
| ELSE                              |
|    { statement 2 }                |
```

Last time we saw the simplest version of the loop:

```
10 X = X + 1
20 PRINT X
30    GOTO 10
```

This is an infinite loop since it has no way to end. One way to end it would be by adding a conditional branch:

```
10 X = X + 1
20 PRINT X
30 IF X < 10 GOTO 10
```

Now, line 20 says that you should jump back up to line 10, ONLY if X is less than 10. (Basic uses the symbols <, >, <=, >= and <> to represent the ideas "less than", "greater than", "less than or equal to", "greater than or equal to" and "not equal to". Try substituting these operations in the fragment above to see their effects.

The IF statement evaluates a conditional statement and then follows one of several paths. You can have a simple IF statement without an ELSE. This says that you want to do something only if the conditional is true, and you have no other statements to process. Since the interpreted Basic IF statement must all fit on one line, we can also use an expanded form when we have several things that we want the statement to perform. We'll see examples of this in the sample program, CHECKBK.BAS.

```
10 ' checkbk.bas
20 ' a simple check book balancing program
30 ' copyright 1987, 1992  s m estvanik
40 '
50 CLS
60 PRINT "Check Book Balancing Program"
70 PRINT
80 INPUT "What is your opening balance";BALANCE
90 PRINT
95 PRINT "Next transaction? (D/eposit, C/heck, Q/uit)"
100 T$=INPUT$(1)
110 IF T$ <> "D" AND T$<> "d" GOTO 210    ' is this a deposit?
120    INPUT "Amount of deposit";DEPOSIT
130    BALANCE = BALANCE + DEPOSIT        ' add to balance
140    PRINT USING "New balance is $#####.##";BALANCE
150       GOTO 90
210 IF T$ <> "C" AND T$<> "c" GOTO 300    ' is this a check?
220    INPUT "Amount of check";CHECK
230    BALANCE = BALANCE - CHECK          ' subtract from balance
240    PRINT USING "New balance is $#####.##";BALANCE
250       GOTO 90
300 IF T$ <> "Q" AND T$<> "q" GOTO 90    ' do they want to quit?
400 PRINT            ' we're done, so show the final balance
410 PRINT USING "Final balance is $#####.##";BALANCE
430 END
```

This version of the program uses only the simple statements that we've discussed thus far. As we learn more about Basic, the exercises will suggest ways you can return to this example to flesh it out. For now, try running this program and follow its operation. (The example programs are included in the tutorial package in a ready to run fashion. You can also clip out segments of code from this tutorial and edit them.)

Since interpreter statements must fit on one line, we're forced to use a GOTO in order to create complex IF-THEN-ELSE statements. The compiler doesn't limit us this way, so we can construct statements without ever using GOTO's.

```
if X > 10 then
   PRINT X; " is > 10"
else
   PRINT X; " is <= 10"
```

[... back to the Contents Table]

2.3 Check Book Balancer

This is the first real program we've looked at, so let's examine it in more detail. One of the goals of structured programming is to make programs that are easy to modify and maintain in the future. To this end, you should include a header at the top of each program that describes what it does, and most important, when it was last modified and by whom. Then in writing the program, use comments wherever they help to explain the flow of the program. Comments are ignored by the Basic interpreter and the compiler. They're indicated by either the word REM or more commonly, the apostrophe (‘). Comments starting with ' may be either the first or last elements on a line.

This allows short comments to be placed more precisely. Combined with a standard indentation comments make the program structure more readable. For example, I indent any statements after an IF statement. I also indent any GOTO that starts a statement. In this way you can see that the statements are dependent on the IF statement and that the control is passed by the GOTO. You'll probably encounter any variants of indenting. The important thing is to be consistent. Remember, though, that indenting is solely for the human reader's benefit. The computer acts on what it reads, not how it's formatted. Improper statements will not improve because of indenting.

If you have multiple IF statements, then the indentations would add up:

```
|    IF { condition A} THEN          |
|       IF { condition B } THEN      |
|          { statement 1 }           |
|       ELSE                         |
|          { statement 2 }           |
|    ELSE                            |
|       { statement 3 }              |
```

In this example, if condition A is true, a second check on condition B is made, resulting in either statement 1 or 2 being executed. If condition A is false, then only statement 3 is executed and condition B isn't even checked.

The check book program first clears the screen (CLS), then announces itself and uses a new command, INPUT, to ask for an opening balance. The INPUT statement prints the string we give it (also called a 'prompt '). It adds a question mark and then waits for the user to enter an answer and hit the enter key. It stores that value in the variable BALANCE. INPUT is simple to use, but does have some drawbacks. Try entering some non-numeric characters. INPUT recognizes illegal characters, but its response is less than elegant. Later we'll see some better ways for handling data input. After the balance is entered, we enter the main loop of the program. The user indicates what type of transaction comes next. (Note that we've changed the structure of our program so that the user can enter deposits and checks in any order.) The INPUT$(1) statement accepts precisely one character from the user and doesn't need a carriage return. This is much friendlier since the user only needs to press one key each time.

Since we don't care whether the user enters an upper or lower case letter, we shouldn't penalize them. Thus we'll allow either when we check. Lines 110, 210 and 300 check to see if the value entered is not equal to one of the valid codes. Thus in line 110, if T$ is not "T" or "t", we go to 210 to check. At any stage, if we find a match, we do the appropriate processing, then go back to get the next transaction.

One last suggestion on GOTO's. Do not jump up in the program, unless absolutely necessary. That is, in reading a program, the normal flow should be downwards. One exception would be a loop such as we have here. Line 90 starts the loop, and at the end of each transaction we return to it.

[... back to the Contents Table]

2.4 Summary

We've looked at some basic ideas of structured programming and created our first useful program. We've also learned the IF-THEN construct. In addition, we've already seen some of the elements of designing user friendly programs.

2.5 Exercises

1. Add a third option that allows the user to enter a monthly service fee. Decide on a code, and add a new transaction that subtracts the fee. Be sure to change the prompt line to show this new code.

2. The current program just ignores bad codes. Add a message line that tells the user that they've entered a bad code, then asks them to reenter their choice.

3. The PRINT USING statements are designed with home use in mind, so they only have 5 figures in the display. What happens if the user inputs a number such as $100,100 ? Assume that any number greater than 10000 is an error. Add an extra check to both the deposit and check section that prevents these numbers from being included. You should print an error message and then go back to the transaction code input line without making any changes to the balance.

4. Only positive numbers are valid for input. Check that each number entered is greater than 0 before allowing it.

[... back to the Contents Table]

3. Loops, Colors and Sound

In the last section, we learned a simple way to have our programs branch on different choices. This time, we'll add more structured loops, color and sound. In the following section, we'll look at arrays and data statements.

3.1 Looping

In the previous program, we used GOTO's to move around. This is discouraged by purists and for good reason. It's too easy to write spaghetti code with meatball logic that jumps all over. This makes it difficult for anyone else to read or understand your code. It's even hard to read your own code after a week or two. To review, if we wanted to add up a random series of 10 numbers given by the user, we could write:

```
10 x = 1
20 sum = 0
30 print x
40 input "enter a number";n
50 sum = sum + n
60 x = x + 1
70 if x < 11 then goto 30
80 print "The sum of the numbers you gave is";sum
```

Basic provides two other ways to accomplish this, both of which make the program more flexible and easier to understand. These are the FOR-NEXT and the WHILE-WEND loops. Using FOR-NEXT, we would write

```
10 sum = 0
20 FOR x = 1 to 10
30 print x
40 input "enter a number";n
50 sum = sum + n
60 NEXT x
70 print "The sum of the numbers you gave is";sum
```

Each FOR must have a corresponding NEXT. The FOR-NEXT statements repeat automatically. The first number is the initial value, the second is the final one. You can also use the STEP command to move by more than one. These loops can be nested as shown in the following example. Note that the last FOR started must be the first one finished.

```
10 input "Starting value (1-10)";x1
20 input "Ending value (20-30)";x2
30 print
40 for x = x1 to x2 step 2
50    for y = 1 to 5
60    print x*y;
70    next y
80 print
90 next x
```

Here we print a series of products, five to a line. The semicolon ensures that all 5 products are printed on a single line. After Y's FOR-NEXT loop is finished, we do a simple PRINT statement to move to the next line.

Another form of looping is the WHILE-WEND pair. Our initial example could be reconstructed as:

```
10 sum = 0
20 x = 1
30 WHILE x < 11
40    print x
50    input "enter a number";n
60    sum = sum + n
70    x = x + 1
80 WEND
90 print "The sum of the numbers you gave is";sum
```

The WHILE statement continues to loop until the condition is achieved. Unlike the FOR-NEXT this might not happen. Consider the following:

```
10 x = 1
20 WHILE x > 11
30    input "enter a number";n
40    sum = sum + n
50 WEND
```

This program contains a common mistake. The value of X never changes, so the loop repeats forever. If you're testing and the program just wanders off and never comes back, infinite loops are prime suspects. Check for non-terminating WHILE loops by inserting a print statement.

In the above examples, a FOR-NEXT loop is actually preferred, since we really intend the loop to be executed a fixed number of times. There are many cases when only a WHILE will do. WHILE's can handle loops when the ending conditions are unclear. For example:

```
10 pts = 0
20 die.roll = int(rnd*6) + 1 ' random number bet 1 and 6
30 while die.roll <> 6 and pts < 20
40    pts = pts + die.roll
50    die.roll = int(rnd*6) + 1
60 wend
70 if die.roll = 6 then pts = 0
80 if pts>0 then print "You made";pts;"points" else print"You lost"
```

This is a simple game in which you have a chance of winning up to 24 points if the "die" manages to avoid coming up 6. RND is a Basic function that returns a random number between 0 and 1. INT(RND*6) then converts this to a random number between 0 and 5. Since dice have no 0's we add one to it. Here the while statement has 2 possible ways to end. Either the die rolls a 6, or the total points becomes 20 or greater. If the loop ends with a die roll of 6, then all points are lost.

[... back to the Contents Table]

3.2 Exercise: Make this game more interactive

After every die roll, show the current points and give the user a chance to quit, keeping their current points. Allow an unlimited number of points to be accumulated. Don't use any GOTO's in your solution.

[... back to the Contents Table]

3.3 Colors

Assuming you have a CGA or EGA card, you can easily add colors and video effects to your programs. There are 8 colors, with 4 possibilities for each color, defined as follows:

	"dark"	"light"	dark, blinking	light,blinking
black	0	8	16	24
blue	1	9	17	25
green	2	10	18	26
cyan	3	11	19	27
red	4	12	20	28
magenta	5	13	21	29
brown	6	14	22	30
white	7	15	23	31

On the CGA you can have any of the 32 colors as a foreground color. This is the color that letters and other characters will use. You can also set any of the 8 colors as a background color. This allows reverse video effects. Try the following:

```
10 color 7,0
20 print "try me"
30 color 0,7
40 print "try me"
50 color 15,1
60 print "try me"
70 color 31,1
80 print "try me"
```

The program included on the disk, COLORS.BAS gives a more extensive display of the color range. First it reproduces the table above, but using actual colors for the numbers. Then it shows all combinations of foreground and background. Note that if foreground and background are the same, the letters can't be read.

Color can be easily abused. Clashing colors or too many colors distract rather than attract. Try to avoid using flashing messages for all but the most important warnings. In particular, don't use flashing colors for input messages. They are difficult to read. Use colors which are easy to read, with the more garish colors saved for special cases. The most readable color combinations on most displays are:

```
|    (bright) white  on blue    |
|    (bright) yellow on blue    |
|    (bright) white  on red     |     (useful for error messages)
|    (bright) yellow on red     |       "      "    "      "
|    (bright) white  on black   |
|             black  on white   |
|    (bright) green  on black   |
|    (bright) yellow on black   |

       (bright) indicates either light or dark is ok.
```

We'll use colors more later when we look at more of the graphics commands. For now, try the following problem:

[... back to the Contents Table]

3.4 Exercise

Add color to the checkbook program given last time. Use the following scheme:

First clear the screen to white on blue using the command:
```
   COLOR 8,1 : CLS
```

For the first prompts, keep the color as white on blue.
If a Debit is chosen, change color to bright yellow on blue.
If a Credit is chosen, change color to bright white on blue.
If the user makes a mistake, change to bright yellow on red.

Remember to change back to the original setting after each special color.

[... back to the Contents Table]

3.5 Sounds

Basic gives 2 methods for making sounds: SOUND and PLAY. SOUND is easier to learn, but harder to use effectively. Its structure is

```
|    SOUND frequency, duration    |
```

Frequencies are in Hertz (cycles per seconds), durations in clock ticks (about 18/second)

```
|   SOUND 220, 18     |  would thus give an A for one second.
```

In practise, you rarely try to duplicate music using SOUND. Instead it's useful for sirens and other sound effects. For example, try

```
| 10 for i = 1 to 10   |
| 20 sound i*100,5     |
| 30 next              |
```

Or, consider the following:

```
| 10 FOR I=1 TO 9 'phaser sounds                          |
| 20 RII!=I/( 90)                                         |
| 30    FOR J= 1000 TO 2700 STEP 200:SOUND J,RII!:NEXT J  |
| 40 NEXT I                                               |
```

Play with the various loop commands to achieve different effects. Different CPU speed may affect these sounds, so they are not generally recommended. Most compilers have a timer function that allows you to delay a set amount of time.

PLAY has a more complex syntax. I'll briefly introduce it here. The best way to learn this command is to practise using it in various ways.

PLAY requires a string that uses Microsoft's music command language. Some of the commands are:

```
| A-G | with optional # or -, play the notes A-G with sharps or flats.

| O n | sets the octave.  An octave goes from C to B.
        n can range from 0 to 6.

| N x | plays note x, from 0 to 84.  This is an alternative to using
      the notes and octaves, but is less useful if you're
      transcribing from musical notation.

| L n | sets the length of the note. Can range from 1 to 64.
      Any value, can be played, even 23rd notes!

| P n | pauses for length n.
```

```
| T n | sets the tempo in quarter notes /minute.  For example
      Largo is 40-60, Adagio is 66-76, Allegro is 120-168.
```

To play a sequence, you just construct a string and give it to PLAY.

```
|    x$ = "efg-fed#"                              |
|    PLAY "T120 L8 o2" + x$ + "p4 o3 l16 " + x$   |
```

We first assign a sequence of notes to x$. Here we're playing E, F, G flat, F, E and D sharp. Then we play this sequence of notes at tempo 120, using eighth notes in octave 2. We pause for a quarter note's time, then move up an octave, change to sixteenth notes and repeat the phrase.

Here's some more interesting music to practise with:

```
| 10 '================== little mathy groves       |
| 20 PLAY "t160 o3"                                |
| 30 PLAY "l8 g4eee4ddg4efe4.d"                    |
| 40 PLAY "gag4a4gab2 p4ga b4b4b4.b g4b4d4.g "     |
| 50 PLAY "o2b4 o3 d4 e4f4g4.ge4d4 o2b4o3d4e2."   |
| 60 X$=INPUT$(1)                                  |
| 70 '===================== rising sun blues      |
| 80 T1$="t200 o2 l4 ge2gb2 o3c+ o3d2 e8d8 o2b2."  |
| 90 T2$="p2 b o3 e2ef8e.d o2b8o3c."               |
| 100 PLAY T1$+T2$                                 |
```

These are some phrases from folk songs. Note that I've used an alternate way of designating the length in line 30. The G4 says to override the default length of 8 for that one note. You can practise by transcribing your favorite music, then changing tempo, or playing style.

In the second example, note that several phrases can be described independently then built up as needed.

[... back to the Contents Table]

4. Arrays and DATA Statements

Last chapter, we added color and sound to our programming kit. We also saw several ways to repeat instructions. In the next chapters we'll finish our introduction by looking at more efficient ways to store data, better methods for performing repetitive sections of a program, and methods for storing data in files.

[... back to the Contents Table]

4.1 Arrays

Arrays are a convenient way of representing groups of data. Consider the months of the year. We could store them in a program as follows:

```
|   MONTH1$ = "JAN"   |
|   MONTH2$ = "FEB"   |
|   MONTH3$ = "MAR"   |
|   MONTH4$ = "APR"   |
|   ...               |
```

Then when we want to print month X we could code:

```
|    IF X = 1 THEN PRINT MONTH1$  |
|    IF X = 2 THEN PRINT MONTH2$  |
|    IF X = 3 THEN PRINT MONTH3$  |
|    IF X = 4 THEN PRINT MONTH4$  |
|    ...                          |
```

This isn't terribly efficient. And what if, instead of 12 items, we had hundreds or THOUSANDS? We might want to get a total of all incomes in a particular group. If there were just a few people to total we could code

```
|  TOTAL.INCOME = INCOME1 + INCOME2 + INCOME3...  |
```

But do you really want to write the equation for 10,000 people? Luckily we have arrays. An array is a grouping of data that still lets us access individual elements. An array is defined by a dimension statement:

```
|  DIM MONTH$(12)  |
```

This defines a group of 13 strings starting with MONTH$(0) and ending with MONTH$(12). What's in each one is still up to the programmer and the program, but now we can access a particular item much easier. Using the months example from above, we still need to define each item or element of

the array :

```
|    MONTH$(1) = "JAN"  |
|    MONTH$(2) = "FEB"  |
|    MONTH$(3) = "MAR"  |
|    MONTH$(4) = "APR"  |
|    ....               |
```

So far no big change, but look at how easy it is to find a particular value. Now we can get month X directly:

```
|    PRINT month$(X)    |
```

X is called an index. If we wanted to print all twelve months, we could use a loop:

```
|    FOR X = 1 to 12    |
|    PRINT MONTH$(X)    |
|    NEXT               |
```

Compare this to the hassle of trying to print all months the first way. When you have larger arrays, the savings become spectacular. These arrays are called singly dimensioned arrays since there is only one index. But we can also think of times we'd like to use several dimensions. Maybe we want to track the production of several product lines over several months. We could set

```
|    DIM PRODUCT(10, 12) |
```

This defines an array that will hold sales information on 10 products for each of 12 months. Thus PRODUCT(5, 11) would hold the sales for the 5th product for the 11th month. Note, we could just as easily defined this as

```
|    DIM PRODUCT(12, 10)  |
```

where we have data for each month for each product. Now the data for the 5th product for the 11th month would be PRODUCT(11, 5). The first method is the one usually preferred. Think of the array as starting with the larger category (here, product type) on the left, moving to subcategories on the right (months).

We can extend the number of dimensions to 3 if we want to show the sales for each day of the month:

```
|    DIM PRODUCT(10, 12, 31)  |
```

In theory you can have 255 dimensions. In reality you'd run out of memory well before using all 255. Each added dimension will raise the required storage by at least a factor of 2. The total memory space that is available to Basic is only 64k. Thus even in our products example, we move from the 11 elements of PRODUCT(10) to the 143 of PRODUCT(10,12) to 4576 for PRODUCT(10,12,31) (11*13*32). Another problem is the fact that most people have trouble conceptualizing more than 3 or 4 dimensions. Usually it's easier to restate the problem. In years of programming I can recall only a few instances where more than 3 dimensions made any practical sense.

Arrays form the basis of most data processing applications, especially in areas like spreadsheets and statistics. In Basic, an array is considered to start from item 0. Many programmers forget about this element and though it will take up a little more space, you often can ignore it too. But there are some cases where it comes in handy. Suppose we have a 5 by 5 array and want to get totals in each direction. Using our product by month example, we'd want to get totals for each product for all months and totals for each month for all products. Without arrays we'd have to construct separate assignment statements for each total. (No, this won't be assigned as an exercise. But just think how long it would take to do this, and how many places you could mistype and cause an error!) The short program total.bas shows this:

```
| 10  ' totals.bas                                  |
| 20  ' do cross totals on an array                 |
| 30  DIM X(5,5)                                    |
| 40  X(0,0) = 0                   ' grand totals   |
| 50  FOR I = 1 TO 5                                |
| 60  X(I,0) = 0                                    |
| 70     FOR J = 1 TO 5                             |
| 80     X(I,J) = RND(1)*10                         |
| 90     X(I,0) = X(I,0) + X(I,J)  ' line totals    |
| 100    X(0,J) = X(0,J) + X(I,J)  ' column totals  |
| 110    X(0,0) = X(0,0) + X(I,J)  ' grand total    |
| 120    NEXT                                       |
| 130 NEXT                                          |
```

```
140 FOR I = 5 TO 0 STEP -1
150    IF I = 0 THEN PRINT "  "   ' extra space
160    FOR J = 5 TO 0 STEP - 1
170    IF J = 0 THEN PRINT "  ";  ' extra row
180    PRINT USING " ###.## ";X(I,J);
190    NEXT
200    PRINT
210 NEXT
```

First we define an array X to be 5 by 5. (Later you can expand this to more rows and columns, but you may run out of space to display it on the screen.) Since we're only going to use the elements that have indices greater than 1, we have 3 classes of elements that would otherwise be wasted. These are all X(i, 0), X(0, j) and the single element X(0,0). We'll use these as follows:

```
X(i, 0) will store the total for row i
X(0, j) will store the total for column j
X(0, 0) will store the grand total of all rows and columns
```

Line 40 sets the grand total to 0. Now we have a double-do loop (not to be confused with the DOO-WAH loop that's often used for timing.) Since this is just a test, we don't want to burden the user by having them enter all the data, so in line 80 we just fill in a random number from 0 to 9. Then we accumulate the line and column totals in lines 90-110. That's all there is to it. To display the results we'll use a pair of reversed loops. This will present the 0 indices in the last rows and columns, so the table will look more like the spreadsheet format you may be used to. In the lower right corner is X(0,0) the grand total.

The PRINT USING statement " ###.##" says to print the value of X using 8 spaces. The value will be shown as up to 3 digits with 2 additional places shown after the decimal place.

[... back to the Contents Table]

4.2 Exercises

1. Change the PRINT USING statement to show only 2 digits with one decimal place. Reduce the total number of spaces used to 6.

2. Why do we need the PRINT USING statement in the first place? Hint: take it out, replacing it with a simple PRINT X(I,J).

3.Change this program to allow creation of a spreadsheet that will produce a table showing 6 different products in columns, with 12 months as the rows. Include labels for each row and col. You can use something simple like "Month 9" and "Prod 2", but should use a loop rather explicit numbered labels.

As you program you'll find countless ways to use arrays. Few programs of any size can get along without them. Even small programs benefit. Let's look at another use. Program PLAYARRY.BAS shown below defines a series of strings.

```
1 ' playarry.bas
10 PLAY "t220"
20 L$(1) = "l8o1 e "
30 L$(2) = "l8o1 g "
40 L$(3) = "l8o1 f "
50 H$(1) ="l32o3 defgab o4 cd"
60 H$(2) ="l32o4 edc o3 bagfe"
70 H$(3) = "l32 o3 dc o2bagfed"
80 H$(4)= "l32 o2 cdefgab o3c"
90 FOR K = 1 TO 3
100 IF K = 3 THEN PLAY "t220" ELSE PLAY "t180"
110 FOR I = 1 TO 12
120 IF K = 1 OR K = 3 THEN PLAY L$((I MOD 3)+1)
130 IF K = 2 OR K = 3 THEN PLAY H$((I MOD 4)+1)
135 ' quit as soon as any key is pressed
140 IF INKEY$ > "" THEN 180
150 NEXT
160 NEXT
170 GOTO 90
180 END
```

[... back to the Contents Table]

4.3 DATA Statements

So far, whenever we've wanted to define initial values, we've just used assignment statements. An alternative is the DATA statement. In its simplest form, it consists of a READ statement and a DATA statement

```
|    READ I  |
|    DATA 5  |
```

When these two statements are executed, I is assigned the value 5. In this case there isn't much savings over the more straightforward

```
|    I = 5   |
```

But the DATA statement is more flexible. You can define a whole array with one concise statement:

```
|    FOR I = 1 to 10 : READ X(I) : NEXT  |
|    DATA 1,3,4,5,6, 6,5,6,5,6           |
```

Otherwise this would take 10 separate statements. You can also make DATA statements conditional. RESTORE declares where the initial data statement begins.

```
| 10 IF X = 1 THEN RESTORE 20 ELSE RESTORE 30 |
| 20 DATA 1,2,3                               |
| 30 DATA 4,5,6                               |
| 40 READ I,J,K                               |
```

Here, when X is 1, I,J,K will be read as 1,2,3 otherwise they will be set to 4,5,6. RESTOREs are useful when you have many DATA statements in a program. Normal usage places DATA statements near the corresponding READ statements, but Basic doesn't care. The program uses the next sequential DATA element.

```
|    DATA 1,2,3                      |
|    READ X, Y                       |
|    ...... [ much more code ]       |
|    DATA 4,5,6                      |
|    READ Z                          |
```

In this fragment, Z is set to 3, since only 2 READ's were done before it. Be careful when coding DATA and READ statements. Having too few data elements will cause a syntax error. Having more data elements than corresponding READ's is allowed, but could cause hard to find errors. A safety measure would be to print out the last of a series of read when testing so that you can ensure that values are being set the way you intended.

Strings in DATA statements cause several additional concerns. When all the strings are single words, they could be read in separated by commas, but if some of them have commas as part of the item, then you'll need to delimit each string by quotes. Thus if we used the statements

```
|    DATA Seattle, WA, Bar Harbor, ME |
|    READ FROM$, TO$                  |
|    PRINT FROM$, TO$                 |
```

we'd see

```
|    Seattle     WA                   |
```

In fact we want the city and state to be paired, so we'd need to write

```
|    DATA "Seattle, WA", "Bar Harbor, ME"  |
```

Let's look at another way to use arrays and data statements. DRAWBOX1.BAS prints boxes on the screen.

First, a slight digression. CHR$() is a special feature of Basic that returns the ASCII equivalent of a number. ASCII is a set of symbols used in programming. The first 128 characters (0-127) are rigidly defined. Some computers such as the IBM PC define an additional set for the numbers 128-255 but these are not standardized. In the IBM system these include the symbols for creating boxes and forms. It's easy to confuse ASCII characters and their corresponding numbers. All characters and numerals are ASCII characters. Thus the upper case 'A' is 65. Adding 1 to it gives ASCII 66 or 'B'. Most confusion comes with numbers. ASCII 49 is the character '1'. Later we'll see ways of using this numeric feature in sorting and alphabetizing. For now all you need to know is that PRINT CHR$(X) will display the X'th ASCII character. Most Basic manuals have an ASCII chart as an Appendix. Another option would be to write a short Basic program that prints all values. The converse of the CHR$() function is ASC(). This returns the ASCII value of a given character. Thus

```
|    PRINT ASC("1")     |
```

would print 49.

```
FOR I = 0 to 255
LPRINT I;CHR$(I)      ' print item and ASCII equiv to printer
NEXT
```

This will end up 5 pages long. Try writing a program that will format this into 8 columns. Thus the first line would have the 0, 32, 64, 96,... elements, the second line would have 1, 33, 65, 97,... You can do this with either one loop and a long print statement or nested loops and a single print statement. Remember that a semicolon (;) after a print line will prevent a skip to the next line. If you use this method, you'll need to have a separate print statement to shift to the next line.

Now we're ready to draw a box:

```
1  ' drawbox1.bas
10  DIM CORNER(4)
20  KEY OFF
30  W = 15 ' width of box
40  H = 6 ' height of box
50  DATA 205, 186, 201, 187, 188, 200
60  READ HORIZ, VERTICAL
70  FOR I = 1 TO 4
80  READ CORNER(I)
90  NEXT
100 X = 0
110 WHILE X < 1 OR X > 25 - H
120    INPUT "Upper left row";X
130 WEND
140 Y = 0
150 WHILE Y < 1 OR Y > 80 - W
160    INPUT "Upper left column";Y
170 WEND
180 COLOR 15,1
190 CLS
200 LOCATE X,Y
210 PRINT CHR$(CORNER(1));      ' top line
220 FOR I = 1 TO W-2 : PRINT CHR$(HORIZ); : NEXT
230 PRINT CHR$(CORNER(2));
240 FOR J = 1 TO H - 2     ' create middle section
250 LOCATE X+J,Y
260 PRINT CHR$(VERTICAL);
270 FOR I = 1 TO W-2 : PRINT CHR$(32); : NEXT
280 PRINT CHR$(VERTICAL);
290 NEXT
300 LOCATE X+H-1, Y
310 PRINT CHR$(CORNER(4));      ' bottom line
320 FOR I = 1 TO W-2 : PRINT CHR$(HORIZ); : NEXT
330 PRINT CHR$(CORNER(3));
340 END
```

First we set the width and height of the box we want to draw. Then we read in values for the horizontal and vertical linedrawing characters Í and º. These are ascii values 205 and 186. Line 60 reads their values and stores them as variables HORIZ and VERTICAL. Next, we need the the four corners:

```
+--   --+
|       |

|       |
+--   --+
```

The ascii values for the corners (201, 187, 188 and 200) are stroed in an array (lines 70-90). We next ask for row and column coordinates. The program draws the upper left corner, a row of horizontal characters and the upper right corner. For each intermediate line, we draw a vertical section, spaces and another vertical. We finish with the 2 lower corners connected by another horizontal line.

[... back to the Contents Table]

4.4 Exercises

1. Allow user to draw several boxes without erasing them You'll need to keep the input on several lines, eg, lines 21 and 22. Clear each line before prompting for new data.

2. Check the ASCII table in the back of your Basic manual. Find the characters that can be used to make a single lined box and add them to the

program. Use a 2 dimensional array. That is redefine the arrays to be CORNER(4,2), HORIZ(2), VERTICAL(2) and then ask the user whether they want a single or double lined box.

3. More involved: This box is drawn from the top down. An alternative would be to draw it as if a pen were sketching it. Rewrite the program to draw the first line, then drop down the right hand side, come back along the bottom right to left, then draw up to the original corner. For the sides you'll need to recalculate the x,y position each time. A FOR-NEXT loop works fine.

We've looked at several different applications of arrays. We've also managed to review many of the constructs that we covered in previous sessions. In the next installments we'll refine these ideas as we start considering how to build larger programs in a structured manner.

[... back to the Contents Table]

4.5 Final Exercise

In preparation for next time, try this as a final exercise:

```
DIM DAYS(12)
DATA 31,28,31, 30,31,30, 31,31,30, 31,30,31
FOR I = 1 TO 12 : READ DAYS(I) : NEXT
```

Write a short program that takes as input a pair of numbers -- month and day. Calculate the number of days elapsed since the start of the year and the number of days till the end of the year.

[... back to the Contents Table]

5. GOSUBS and FUNCTIONS

This time we'll be examining GOSUB's and FUNCTIONS. Up to now, when we've wanted to repeat a section of code, we've had 2 choices. We could just copy the code over, or we could set up a loop. Sometimes that still leaves us with a messy solution.

5.1 Why GOSUBS?

In the previous chapter, one of the exercises involved writing a short program that took as input a pair of numbers -- month and day, then calculated the number of days elapsed since the start of the year and the number of days till the end of the year.

But what if we want to input 2 sets of numbers? One way would be to repeat the code. Another would be by using a GOSUB. This is a special command that tells the program to jump to a particular line. It differs from a GOTO in that when the command RETURN is found, the program jumps back to the statement that called it. This lets a whole section of code be used in several places. Thus

```
10 X = 1
20 GOSUB 100
30 X = 2
40 GOSUB 100
50 X = 3
60 GOSUB 100
70 END
100 PRINT "X squared is";X*X
110 RETURN
```

This will produce 3 lines showing the squares of 1, 2 and 3. Note the use of the END statement. Take it out and see what happens.

We can turn the elapsed days calculation into a GOSUB as follows:

```
10 'julian.bas
20 DIM DAYS(12)
30 DATA 31,28,31, 30,31,30, 31,31,30, 31,30,31
40 FOR I = 1 TO 12 : READ DAYS(I) : NEXT
50 INPUT "Start month";M
60 INPUT "Start day";D
70 GOSUB 200
80 E1 = ELAPSED
90 INPUT "End month";M
100 INPUT "End day";D
110 GOSUB 200
120 DATE.DIF = ELAPSED - E1
```

```
130 PRINT
140 PRINT "difference =";DATE.DIF
150 END
200 '===================== calc elapsed time in days
210 ELAPSED = D
220 FOR I = 1 TO M - 1
230 ELAPSED = ELAPSED + DAYS(I)
240 NEXT
250 PRINT ELAPSED;"days elapsed. ";365-ELAPSED; "days to go"
260 RETURN
```

Here we ask for 2 sets of numbers and then calculate the difference between them. When working with dates for comparison you'll usually want to deal with the number of days past the beginning of the year, so this conversion routine is quite handy. This notation for dates is called Julian (as in Caesar).

[... back to the Contents Table]

5.2 WE INTERRUPT THIS PROGRAM FOR A POLITICAL ANNOUNCEMENT...

Basic is often criticized for its lack of structure and difficulty in reading and maintainence. This is more of a programmer's problem though, not the language's. You can write structured programs in Basic just as you can write spaghetti code in Pascal or unmaintainable trash in C. Structured methods will produce good code in any of the languages. If you follow these guidelines you'll be able to keep your programs readable and easy to maintain.

** The main portion of the program should start at the top and proceed to the bottom. This means using FOR-NEXT and WHILE loops rather than GOTO's. (In older books or articles you may see the suggestion to jump to the end of the program for initializing or other reasons. This made a small amount of sense in the old days, but modern Basic interpreters are much more efficient and you won't notice any difference. What you will get is a program that's harder to work on.)

** Whenever possible, use GOSUB's and, once we've defined them, function calls to recycle repetive sections of programs. (Once a GOSUB is working you'll often be able to use them in other programs. Functions are even more transportable.) With a compiler, we'll use subroutines rather than GOSUB's for even more modularity.

** Start each GOSUB with a comment line. This should be the ONLY entry to a GOSUB. It's legal to enter a GOSUB at any point, but it's just asking for trouble. There are a very few instances where it's justified, but I'd put it at less than 1%. Using the comment line helps to delineate GOSUB's in the program.

** Similarly, the last statement in any GOSUB should be the only RETURN. By applying these two rules you abide by one of the central dogmas of structured programming -- single input, single output. This ensures that every time the subroutine is used it's used the same. Thus you won't get unpredictable results. Sometimes you'll have gosubs that finish in the middle of a section. Rather than succumb to the temptation to RETURN from that point, use a GOTO to jump down to the single RETURN at the bottom. This is an insignificant increase in the amount of code, but results in an immense increase in readability and ease of maintainence. It also makes it easier to trace and isolate problems. If you know there's only one entry and one exit, you can concentrate on the routine that's causing the problem. You won't have to worry what conditions are where the gosub is called. If there were multiple entries or exits this would be an additional problem.

```
            GOSUB 100
            ....
            END

    100     ' ----------- subroutine to do stuff...
            { compute stuffs }
            ' are we done?
            IF { final condition achieved } THEN 200
            { compute more stuffs }
    200     RETURN
```

Stringing these ideas together we can draw a prototype program:

```
    ' main program
      GOSUB 100     ' init stuff

    ' process stuff
```

```
|       FOR X = 1 to whatever                          |
|       GOSUB 200     ' first part of processing       |
|       GOSUB 300                                      |
|       NEXT                                           |
|                                                      |
|       GOSUB 400     ' final stuff                    |
|       END                                            |
| 100   '=============== init                          |
|                                                      |
|       { .... do processing here }                    |
|                                                      |
|       RETURN                                         |
| 200   '=============== first processing              |
|                                                      |
|       { .... do processing here }                    |
|                                                      |
|       RETURN                                         |
| 300   '=============== second processing             |
|                                                      |
|       { .... do processing here }                    |
|                                                      |
|       RETURN                                         |
| 400   '=============== final stuff                   |
|                                                      |
|       { .... do processing here }                    |
|                                                      |
|       RETURN                                         |
```

This is a common way to design a relatively straightforward program. It's much to be preferred to flowcharts. Here we've outline the structure of our entire program without getting bogged down in petty details. Another good feature of this design is that we can prototype. We could write the initializing section and the final processing section without worrying for the moment about the 2 middle sections. We could just put some print statements in there to alert us to the fact they need to come later. This method, called Top-Down Design starts with the most important elements of the program, the overall structure and works in ever more detail. In this way, the interactions among the program parts is being tested from the start. Try this method with simple programs and you'll find that soon you can tackle more complicated projects. If you're interested in more on structured programming, look for books by Constantine or Yourdon. Most of the ideas in these books can be applied to any language.

BACK TO OUR SCHEDULED PROGRAMMING....

[... back to the Contents Table]

5.3 Renumbering (RENUM)

If you're using a compiler, you never need to worry about renumbering. This feature alone might be enough to convince you to acquire a compiler.

RENUM is a command in the Basic interpreter that lets you renumber your program. It's especially useful since it takes care to maintain any references you've set up. All GOTO, RESTORE, IF-THEN and GOSUB relations will be the same after the RENUM as they were before. The numbers may be different. This is the preferred way of changing numbers. You can do it by hand, but if you miss one, it'll be hard to find. The simplest form of RENUM is just

```
|  RENUM                      |
```

This renumbers the entire file, using 10 as an increment. If you have a large file and want to reorder just part you can use

```
|  RENUM [newnum], [oldnum]   |
```

This will start at line [oldnum] and change it to [newnum] and continue renumbering from there. Try this on a practise file until you feel comfortable with the power of this command.

There are 2 philosophies about renumbering. Some people only use the RENUM from the start. Here their GOSUB's will often have changed numbers. As long as you keep current listings, there's no problem. Others design a program with large separations between gosubs. Thus you might define them as 1000, 2000, 3000, ..., and interpolate later as necessary. This method keeps the subroutines in the same place, but is more tedious to renumber. You'd need to issue several commands:

```
|    RENUM 1000,1000                          |
|    then list to find what used to be 2000   |
|    RENUM 2000,[new 2000 place]              |
|    then list to find what used to be 3000   |
|    RENUM 3000,[new 3000 place], etc         |
```

Try this with a file with several GOSUB's. The first method is certainly the easiest, and if you're careful to follow the guidelines above your program will still be readable.

[... back to the Contents Table]

5.4 Exercise

A. Design a program that will take 2 dates, prompting for year, month and day. Then calculate the number of days that separate these two dates.

Hint: in our first treatment we ignored leap years. But when you know the year, and are looking over several years you don't have that luxury. You'll need to include a check for leap year in the new elapsed days gosub.

[... back to the Contents Table]

5.5 FUNCTIONS

Functions are similar to gosubs in that they provide a means of using the same code in multiple ways and places. The difference is that functions are defined at the start of a program, and, in interpreted Basic, are limited to single lines. The other difference is that there can only be one value returned from a function. The format for defining and using a function is:

```
|   DEF FNstuff(x) = { whatever the functions going to do }  |
|   ......                                                   |
|   Y = fnstuff(x) + 10                                      |
|   .....                                                    |
|   if fnstuff(x) then print "text 1" else print "text 2"    |
```

Note that a function can be used anywhere a variable might be on the right side of an assignment statement or in a conditional. You can't use a function on the left hand side though.

Actually we've already used several functions. RND is a function, so is CHR$() that we've used to print ascii characters for the boxes. These are system functions since they come as part of the language. Other useful system functions include STRING$() and SPACE$(). Here's some examples:

```
| 10 for i = 1 to 10                                          |
| 20 print SPACE$(i);i                                        |
| 30 next                                                     |
| 40 for i = 0 to 255                                         |
| 50 if i<10 and i>12 then print string$((i mod 50) + 1, i)   |
| 60 next                                                     |
```

[... back to the Contents Table]

5.6 Short exercises

Why do we use (i mod 50) + 1 instead of just i? why do we need the +1? Why don't we print characters 10 to 12? Try taking these out and see why.

Three very useful functions are Left$(), Right$() and Mid$(). These allow us to process parts of strings. Try the following:

```
| 10 x$ = "abcdefghijklmnop"  |
| 20 print left$(x$, 5)       |
| 30 print right$(x$, 5)      |
| 40 print mid$(x$, 5,5)      |
```

Left$ and right$ give the left- and right-most characters. The length to be used is given as the second parameter. Mid$() is more versatile. MID$(x$, x, y) returns y characters, starting at the x'th. A final function for now is LEN(). This returns the length of a string. Thus if we want to copy all but the last 2 characters of a string, we could write

```
|  Y$ = left$( x$, len(x$)-2) |
```

We don't even have to know how long x$ is. We should check that x$ is at least three long, though.

[... back to the Contents Table]

5.7 More Exercises

1. Write gosub that strips blanks from the end of a string x$. print the

original length and the new length of the string.

2. Write a gosub that strips multiple blanks from a string, reducing them to single blanks. Strip all trailing blanks.

Writing your own functions is straightforward. All functions must start with DEF FN then up to 6 characters to name the function.

```
| DEF FNMAX( a,b) = abs( a >= b ) * a + abs( b > a ) * b |
```

Here's another twist on using conditionals. The phrase abs (a >= b) translates to 0 or 1 depending on the values of a and b. So, if a is larger than or equal to b, we'll return

```
|  (1 * a) + (0 * b)  |
```

Similarly,

```
| DEF FNMIN( a,b) = abs( a <= b ) * a + abs( b < a ) * b |
```

Some guidelines in designing functions: if a variable appears in the definition itself, then it can be replaced by whatever is used in calling it. Any other variables used in the function take the value of the current state of that variable. Thus

```
|     x = fnmin( x, 100)  |
```

will ensure that x is no bigger than 100. The values X and 100 are sent to replace a and b. If we have the function

```
|     DEF FNMAX2(a) = abs( a >= b ) * a + abs( b > a ) * b  |
```

then we'd call it with

```
|     x = fnmax2(x)   |
```

and the function would use whatever value b is currently set to. This can cause strange happenings in your programs. It's legal, but not recommended as good technique.

We can even combine functions in defining a new function:

```
|  DEFMINMAX(min, max, x) = fnmin( fnmax( x, min), max) |
```

Does this make sense to you? If not, try to work it through using a call like

```
|   X = fnminmax(10, 100, X) |
```

This is a useful function that ensures that X is in the range between 10 and 100. The function first takes the maximum of X or min, then takes the minimum of x or max. Use this in programs like the checkbook program where you can define the expected maximum and minimum values.

Function writing is some of the most fun in Basic. You can be quite creative. While normally obscure tricks are frowned upon, in functions, they're almost okay, as long as you comment them. Once they work, you can use them in many different places. Another advantage, is that, with proper naming conventions your mainline code will be self-documenting. Which of the following is easier to understand?

```
| X = abs( a <= b ) * a + abs( b < a ) * b |
```

or

```
| X = fnmin(a,b)                           |
```

Do it once. Comment it. Then put it away and call it when needed.

[... back to the Contents Table]

5.8 Exercise using GOSUBS or FUNCTIONS

Write a routine that accepts a string and returns a string with 10 @'s before and behind it (@ is ascii 64). Put a space before and after the name. Thus if we send the routine x$ = "Steve" we'll get back

```
|   "@@@@@@@@@@@@ Steve @@@@@@@@@@@@" |
```

[... back to the Contents Table]

5.9 Extra Credit

Here's a project that will also give you an idea of the way spreadsheets work:

This project is more difficult than most we've looked at. Even if you don't do the actual programming, it would be worthwhile to design a prototype on paper so that you understand the methods involved.

Start with the modified totals program from last time which shows 6 products over 12 months.

1. Write a 2 functions that take as input the I,J coordinates of the array X and return the screen row and column where that element should be printed.

```
|   DEF FNROW(i,j) = ???                  |
|   DEF FNCOL(i,j) = ???                  |
```

Now when we want to update element X(I,J) we can write

```
|   ROW = FNROW(I,J)                      |
|   COL = FNCOL(I,J)                      |
|   LOCATE ROW, COL                       |
```

In fact, we can eliminate the assignments and just use the functions:

```
|   LOCATE FNROW(I,J), FNCOL(I,J)         |
```

2. Move the totals calculations to a GOSUB.

3. Use a gosub to prompt on lines 23 and 24 for the row and col to update. Check that the row and columns are valid. Give an error message if they're not. Don't allow entry of 0's as we're going to calculate them.

4. Once you have the new row and column, prompt and get a new value, then recalculate the totals and use the functions to redisplay only the fields that have changed.

One approach would be to get the new row and col I,J then store the old value of X()

```
|   OLDVAL = X(I,J)                       |
```

Now recalc the totals:

```
|   X(0,J) = X(0,J) - OLDVAL + X(I,J)     |
|   X(I,0) = X(I,0) - OLDVAL + X(I,J)     |
|   X(0,0) = X(0,0) - OLDVAL + X(I,J)     |
```

Then redisplay these 3 values plus X(I,J)

In outline form:

```
|     { get initial values }                                  |
|     GOSUB 1000              ' calc totals                   |
|     { display initial values using functions                |
|       to locate the elements }                              |
|                                                             |
|     WHILE {still asking for more changes}                   |
|       GOSUB 2000              ' get new row, col            |
|       GOSUB 1000              ' calc totals                 |
|       { use functions to display totals & new value}        |
|     WEND                                                    |

| 1000 '======= calc totals                                   |
|      RETURN                                                 |

| 2000 '======= get new row & col                             |
|      RETURN                                                 |
```

[... back to the Contents Table]

6. Files

In previous chapters we learned how to structure programs and reuse sections of code with GOSUBs and FUNCTIONs. At this point, you've learned enough about Basic to write useful programs. However, we still have no way of preserving the results of a program. What if we want to keep results from

one run to another. The solution is the use of files.

We've already used files to save our Basic programs. When you do a normal save from within Basic, only the Basic interpreter can use it. If you want to see the program in more readable form, you can save it as an ASCII file by modifying the save command:

```
| SAVE "progname.bas", A   |
```

Try this with one of your programs, using a different name so that you have the original and the ascii version. Then exit Basic and use DIR to look at the sizes of the programs. Notice that the ascii cersion takes up more room. Next use the TYPE command to examine the files:

```
| TYPE "origname.bas"      |
| TYPE "progname.bas"      |
```

If you have an editor or word processor, you'll find that you can modify the ascii version but not the original. (If you use an editor, though, you'll be responsible for putting in your own line numbers.)

Other common uses of files are to hold data that must remain when the computer is turned off, or to handle large quantities of data. Basic provides 2 ways to handle files -- sequential and random access.

[... back to the Contents Table]

6.1 Sequential versus Random Files

Some of this chapter may seem more techincal than previous installments. Don't worry about the details. The important thing is to understand the distinctions between the 2 types of files and when each is appropriate.

All files contain records. Records are the repeating elements within a file. They in turn are usually broken down into fields. For example, a file record for a mailing list program might have fields with the name, address and phone number of each person. Sequential and random files handle records and fields quite differently. Each has definite advantages and drawbacks.

Sequential files read or write from the start of the file and proceed in an orderly fashion to the end of the file. Think of them as a cassette tapes. In order to find out what's in the tenth record, we need to read the preceding 9 records. (We may not do anything with the information we read, but we have to read it). Random files give you immediate access to any record in the file. A jukebox with its dozens of records is a good example. When you request a record, the mechanical arm goes directly to the record you requested and plays it. In computer programs, sequential files must always be accessed from the beginning, while random files can select any record at any time.

[... back to the Contents Table]

6.2 What's in a file?

Consider a file as a group of bytes. Any additional structure is our logical description for convenience and understanding. We'll set up a simple data record and see how the two types of files deal with it. Our data consists of the following fields:

```
| name  |
| city  |
| age   |
| phone |
```

The following program asks for the information to make 3 records, then stores them to a file. It then reads that file and displays the results on the screen. Ignore the actual commands for the moment, and concentrate on what the program is trying to do. First we'll do it sequentially. (see the program SEQFILE.BAS)

We read the name, city, age and phone number (lines 30-80), then write them to the file (lines 90-120). When we're done, the file physically looks like this:

```
| ------------------------------------------------------ |
| .name............./.city......./age/.phone.../.name.|
| ...../.city......./age/.phone.../.name......./.city.|
| ...../age/.phone...                                  |
| ------------------------------------------------------ |
```

However, when we read it back in, it's interpreted this way:

```
|    .name............/     |
|    .city......./           |
|    age/                    |
|    .phone.../              |
|    .name.. ...../          |
|    .city......./           |
|    age/                    |
|    .phone.../              |
|    .name....../            |
|    .city....../            |
|    age/                    |
|    .phone...               |
```

Note that each record takes up a variable amount of space in the file. Thus we have no way of predicting where a particular record begins. If we start at the beginning and just read one record, field by field that never concerns us. Sequential files are thus most useful when we have either very short files, or when we know we'll always want to read the entire file into memory. Their main advantage is that they're easy to program and maintain.

We'll look at the organization of a random file, then return to see how to program them. The following program performs the same tasks as the first, but creates a random access file (See RANDFILE.BAS)

Again, we read the name, city, age and phone number, and write them to the file. This time, the file will physically look like this:

```
|   ---------------------------------------------------  |
|   | name................city.............agphone...|  |
|   | name................city.............agphone...|  |
|   | name................city.............agphone...|  |
|   ---------------------------------------------------  |
```

Even though the file is called random, the data appears ordered! Each record consists of 48 bytes or characters. The information is stored exactly the same for each record. Thus if we want to find the third record, we know that it begins on the 97th byte. There's no need to look at the intervening information. We can point directly to the start of the record and read it. Random access files are a bit more complex than sequential, and take more programming effort to maintain, but they are much more flexible. They're best suited to cases where data will be required in no particular order.

Random files are also preferred for large files that are frequently updated. Consider, if you have 1000 records and change 10 of them, a sequential file makes you read all 1000, make the changes, then write all 1000 again. That's 2000 disk reads and writes. With a random file, you only need 10 reads and 10 writes.

[... back to the Contents Table]

6.3 Using Files

Now that we have some idea of why we have two types of files, let's look at how to use them:

The programs introduce several new commands and concepts. The first is the control of files. The OPEN and CLOSE commands tell DOS which areas of the disk to manipulate. You can have several files open at one time. For example, you might have a customer file, an invoice file and a pricing file. Basic distinguishes among them by assigning a number to each one. The # sign in front of the file number is optional, but recommended to distinguish it as an identifier rather than a numeric value. OPEN is used by both types of files, but has a different syntax for each. CLOSE is used when you're finished with a file. When you leave a Basic program, all files will be automatically closed, but it's good practise to have your program do it.

[... back to the Contents Table]

6.4 Sequential Details

Sequential files must be opened for either reading or writing. In either case, an invisible pointer is maintained that indicates where the next record is coming from. The format of the statement is

```
|   OPEN filename FOR [ INPUT / OUTPUT / APPEND ] AS handle  |
```

where filename is any explicit filename or variable. Usually existing files are read first, new information added, then the entire file is written out. APPEND is a special form of OUTPUT. Any existing file is kept and new information is added to the end of the file. An example might be a file that keeps a list of errors encountered during the program. There's no need to read in previous errors, but you also don't want to destroy them. So you

use APPEND to add to the back of the file.

Since the filename can be a variable, you can make your programs more user-friendly by showing the user what files are available. For example, if you have a series of files that are called MAR.DAT, APR.DAT, etc, you could display a list with the FILES command and then prompt for the one the user wants. The FILES command puts a directory on the screen. You can use wildcards to limit the files displayed:

```
|      10 FILES "*.dat"                            |
|      20 INPUT "Which file to report"; filename$  |
|      30 OPEN filename$ FOR OUTPUT AS #2          |
```

Records are written and recovered with the PRINT #, WRITE #, and INPUT # statements. INPUT # reads the requested variables from the given file. You have to be careful that the variable types match. Ie, you can't read a string into an integer variable. PRINT # has several problems for beginners, mostly related to how it formats the fields before writing. The Basic manual describes the problems and their solutions if you're in a masochistic mood. For our purposes, the WRITE # statement is more useful. It encloses each string in quotation marks and separates fields on a line by commas. In this example, we've made it even simpler by writing each field separately. (WRITE places a linefeed at the end of each operation, so when you TYPE TEST.DAT each field will be on a separate line. This uses an extra byte per line, but makes data files easier to display and programs easier to debug.)

[... back to the Contents Table]

6.5 Notes on Random Files

Random files, as illustrated in the second program use the same OPEN statement for input and output. After assigning a number, they also require a length for each record. The FIELD statement is also required before a random file can be used. This command defines the length of each field in a record. You should be careful to use different variable names for the field elements and for your actual data. There are some places where Basic lets you use the same name, but you're taking an unncessary risk of causing bugs. Since the field statement defines the length of each field, you have to do a little more planning with random files. In the sequential file, we never had to consider the length of a name, or whether a phone number had an area code attached. Here, we make the conscious decision that names will be only 20 characters, and that phone numbers will not have area codes. A special case is the storage of numbers. All items in random files are stored in a special format. For numbers, this means all integers are stored as 2 bytes. Even if the actual number is 4 or 5 digits, it's compressed before storage using the functions defined below. (Thus random files are often much smaller than sequential files.)

The next difference is seen in the method used to write a record. First all fields are determined. Then, each element of the field statement is set up using the LSET statement. For strings, this is just an assignment:

```
|      LSET field.element = string.variable         |
```

For integers, we first need to make the number into a string using the special function MKI$()

```
|      LSET field.element = MKI$( integer.variable) |
```

(Similar MK functions are available for converting single and double precision numbers.)

Then we write this record with the statement

```
|      PUT #1, rec.number   |
```

Note that we don't even mention the fields. The FIELD statement automatically takes current values of n$, c$, a$ and p$ associated with file #1 and uses them in GET and PUT statement. Note also, that if, for the next record, we just changed the n$, the previous values of the other elements would remain.

To read a record, we need only it's position. To read the third record:

```
|      GET #1, 3          |
```

Now the element values are in n$, c$, a$ and p$ so we need to translate them to variables that can be used in our program (lines 190-220). The converse of MKI$() function is CVI(), or convert integer.

That's it! Now we have 2 methods for saving data and several ways to manipulate the files.

[... back to the Contents Table]

6.6 Questions and Answers

*** Add verification to the programs to ensure that the age is within a certain range and that all phone numbers are of the form XXX-XXXX.

For the following, write down what you think will happen before trying it with the programs.

*** What happens if you enter unanticipated data to each file? Eg, what occurs if you enter a 3 or 4 digit age? or a phone number with zip code?

*** What happens if the name contains commas or quotation marks?

*** What happens if you open and read a random file in a sequential manner or vice versa?

[... back to the Contents Table]

6.7 Projects

These projects are a little longer than the average, but most of them use sections we've done previously. When you finish these you'll have a good working understanding of Basic files.

1. Sequential files
Use the earlier checkbook programs to create a checkbook file. This should use the following records:

```
check number
description
amount
```

At the start of the program, you'll need to read in the starting balance. At the end, you'll have the ending balance. An outline of the program might be:

```
dim check(200), desc$(200), amount(200)

open "checks.dat" for input as #1
input #1, start.balance
input #1, n.checks
for n = 1 to n.checks
input #1, check(n), desc$(n), amount(n)
next
input #1, final.balance

{ prompt for transactions, keep a running total }
close 1
open "checks.dat" for output as #1
write #1, start.balance
write #1, n.checks
for n = 1 to n.checks
write #1, check(n), desc$(n), amount(n)
next
write #1, final.balance
close
```

It would also be nice to have a report program. This would just read the file and print a report. You should have the deposits and debits in two different columns.

2. Random files
Change the earlier name and address example to allow updates and additions. In outline form:

```
open "test2.dat" as #1 len=48
field 1, .....

prompt for highest record number
[ normally this would be stored in the file itself,
  but for simplicity, assume that the user must know. ]

prompt "Add or Update or Display?"

If {ADD} then
    {record number?}
```

```
|           {prompt for information}                                  |
|           {write record}                                            |
|                                                                     |
|       If {Update} then                                              |
|           {record number?}                                          |
|           {read current information}                                |
|           {display current information}                              |
|           {information to change?}                                  |
|           {write record}                                            |
|                                                                     |
|       If {Display} then                                             |
|           {record number?}                                          |
|           {read current information}                                |
|           {display current information}                              |
```

Note that there are several candidates for GOSUBs or FUNCTIONs here. What happens if you read a record beyond the end of the file? What if you write past the end?

If you don't have the time to do the entire program, implement only one part of it. Put in the prompting for the other sections anyway. If those sections are selected, then print a short message saying that the selection chosen isn't available yet. This is a method that's often used in actual software development. You block out the main segments of the program, then use stubs to indicate where later functions will be. This way a partial program can be tested early rather than trying to debug an entire program at once.

[... back to the Contents Table]

7. Simple Graphics

In the next few sections, we'll look at the graphics abilities of the IBM PC. To fully use these chapters, you'll need access to an IBM with a CGA, EGA or VGA board and monitor. These are graphics adapters that allow you to go beyond simple text.

[... back to the Contents Table]

7.1 WIDTH

So far whenever we've written information to our monitor screen, we haven't done anything special. Thus we've accepted Basic default modes of text screens with 80 columns. There are several ways we can change these defaults. We can set the width of a text screen to be either 40 or 80. In 40 column, the letters are larger, so sometimes easier to read. 80 column mode is crisper with sharper colors. Both modes have their applications. To change from one mode to another, use the WIDTH command

```
|       WIDTH screen.width    |
```

On a monochrome screen, the width command works, but the size of characters doesn't change. All that happens is that display is limited to the left hand side of the screen.

[... back to the Contents Table]

7.2 SCREEN

Let's examine how the IBM screen display is set up. This is specific to the IBM PC environment. If a machine fails the compatibility test it's often related to how its video display is set up. Any "100% compatible" machine must be able to perform all the commands that we'll discuss in these next few sections. (You can use the example programs to test their claims when shopping!)

The following few paragraphs may tell you more than you want to know about the internals of video displays. You can skim them if you wish, then catch up with us below (just press F2)

The first thing we'll need to understand is what an attribute is. Attributes describe how a character is displayed on the screen. We've actually been using attributes without worrying about them up to now. Basic lets you do this with the COLOR command. When you enter,

```
|       COLOR 15,1         |
```

you're telling Basic to set the attribute to 31 which is displayed as intense white on a dark blue background. COLOR commands stay in effect until the next COLOR command is issued. But why 31? The attribute byte, like all bytes can take values from 0 to 255. We can look at it as a series of 8 bits, each of which can be 0 or 1. The positions within the byte are interpreted as follows:

```
|    +--+--+--+--+--+--+--+--+                          |
|    |  |        |  |  |     |                          |
|    +--+--+--+--+--+--+--+--+                          |
|      \  \_____\  \  \______\                          |
|        \      \      \      \                         |
|         \      \      \      \     foreground = 000 to 111  |
|          \      \      \     intensity 0 or 1        |
|           \      \     background 000 to 111          |
|            \     blinking 0 or 1                      |
```

Looks intimidating, but in practise, it's easy to use. It's an extremely efficient way to code the foreground color, its intensity, the background color, and whether or not the character is blinking, all in one number. The foreground color can vary from 0 to 7, in binary terms this means

```
|      000 = 0   black              |
|      001 = 1   blue               |
|      010 = 2   green              |
|      011 = 3   cyan               |
|      100 = 4   red                |
|      101 = 5   magenta            |
|      110 = 6   yellow/brown       |
|      111 = 7   white              |
```

Thus any 3 bits can be used to code for 8 numbers. Next we code the intensity by setting the 4'th or 8's bit on if we want intense color, leaving it at 0 otherwise. So to code for intense white, we'd use 1111. The initial 1 says intense, the next 3 give 7, white's code. Binary 1111 is the same as decimal 15, the number we've been using for intense white all along. We can also code 8 possible background colors using another 3 bits. Since these start in column 5, this is the same as multiplying them be 2 to the 4th power = 16. (Column 1 is 2 to the 0th = 1.) That brings our total to 7 bits. We use the last, the 128's column bit to show whether or not to blink. We've wasted nothing and stored 4 pieces of information in one tiny byte. There's no need to remember all the details. To use attributes you need only use the following formula:

```
|  attribute = 16 * background color + foreground color   |
|  if intense then attribute = attribute + 8              |
|  if blinking then attribute = attribute + 128           |
```

We can make this into a function:

```
|  DEF FNATT(fgd, bgd, intense, blink) =                  |
|      128 * blink + bgd*16 + intense * 8 + fgd           |
```

so if we wanted to calculate the attribute for intense white on blue, we'd use:

```
|      att = fnatt(7,1,1,0)              |
```

Okay, the skimmers should have caught up with us by now....

In order to store information on the screen, we need to know what to put there, and how to display it. The IBM method stores the information as repeating pairs of bytes. Even numbered bytes tell which ascii character, odd bytes give their attribute.

If each character needs 2 bytes, then a 25 line screen of width 80 requires 4000 bytes. A 40 column screen needs only 2000. However, the IBM video display has room to store 16K! Early applications failed to capitalize on this extra memory, since the tricks we'll look at now work only with color graphics adapters and their successors. It turns out that we can use that extra memory to display multiple screens at once. We can think of the IBM memory as being a chunk of 16K broken down as follows:

```
|    -------------------------------   |
|    |   80 col    |    40 col    |   |
|    -------------------------------   |
|    |0            |0             |   |
|    |             |----- 2K -----|   |
|    |             |1             |   |
|    ----- 4K -----|------4K------|   |
|    |1            |2             |   |
|    |             |----- 6K -----|   |
|    |             |3             |   |
|    ----- 8K -----|----- 8K------|   |
|    |2            |4             |   |
|    |             |---- 10K -----|   |
```

```
|     |              |5              |    |
|     ---- 12K -----|---- 12K------|    |
|     |3             |6              |    |
|     |              |---- 14K -----|    |
|     |              |7              |    |
|     ---- 16K -----|---- 16K------|    |
```

Normally, we'd say that Basic limits us to 64K of code and data in our programs. However, the IBM video display can be thought of as an additional 16K of memory. We'll look at some ways that extra memory can be used. The first and simplest way is just to display information. In the default screen of 80 columns and 25 rows, we have 4000 bytes of information. Looking at the map, we see that in fact there's room for 4 pages of information, numbered 0 to 3. Similarly, the 40 column width gives us 8 pages, from 0 to 7. The SCREEN command lets us switch among these pages.

```
|      SCREEN mode, burst, apage, vpage  |
```

For now, the only mode we'll use is 0 which is text mode. The burst is 0 for RGB screens, 1 for composite. (This is an archaic leftover from early video monitors. Set the burst to 1 for any modern system.) The two interesting guys are apage and vpage, standing for active page and visual page. These can range from 0 to 3 or 7 depending on width. The visual page is the screen you're currently showing on your monitor. The active page is the page to which your program reads or writes. Normally these are the same, but some interesting effects are possible if you vary them. Screens are easier to show than explain. (See program SCR.BAS).

This program shows how screens work. First we write a line on each of the 4 screens, switching both active and visual. Then we write to each of the screens, while keeping 0 as the active screen. Finally we switch from screen to screen without writing anything more, to prove that in fact we have done something. Why bother? What happened while the program was writing to the other pages? Did you notice a delay? What if you had several pages of information, such as instructions that you wanted to store for easy reference, and didn't want to rewrite each time? If you stored them to an alternate page, you'd only have to write them once, then by just shifting screens you could get that information instantly.

[... back to the Contents Table]

7.3 Exercise

Using the checkbook balancing program, change the program so that any errors are displayed on an alternate page. Write the error first, then shift to the page. Keep track of errors and write each one on a new line. (Be careful that you don't try to write past line 25!) After each error, shift to the page showing accumulated errors, then wait for a keypress before coming back to the main program.

The commands we'll be learning next are:

```
|     CIRCLE     |
|     LINE       |
|     PSET       |
|     PAINT      |
```

In addition we'll see new uses for:

```
|     COLOR      |
|     SCREEN     |
```

We'll look at 2 programs this time that illustrate these commands:

** PALETTE.BAS shows the combinations of colors we can achieve

** LINES.BAS shows some of the straighter applications

First we'll look at circles and colors:

```
| 10     'palette.bas                                          |
| 15    KEY OFF                                                |
| 20      CBK = 1                                              |
| 30     PALET = 0                                             |
| 40     P = 0                                                 |
| 50     P2 = 0                                                |
| 60     SCREEN 1,P2 : COLOR CBK, PALET : CLS                  |
| 70     GOSUB 250                                             |
| 80     X$=INPUT$(1)                                          |
| 90    WHILE X$ <> " "                                        |
```

```
100       IF X$ <> "P" AND X$ <> "p" THEN 160
110            P = P + 1
120            IF P > 2 THEN P = 0
130            PALET = P MOD 2
140            IF P = 2 THEN P2 = 1 ELSE P2 = 0
150            SCREEN 1, P2
160     IF X$ <> "B" AND X$ <> "b" THEN 190
170          CBK = CBK + 1
180          IF CBK > 31 THEN CBK = 0
190     COLOR CBK, PALET
200     GOSUB 250
210     X$ = INPUT$(1)
220  WEND
230  SCREEN 0,0 : COLOR 15,1
240  END
250 ' --------- showit
260  FOR I = 1 TO 3
270  CIRCLE (50 + I* 50, I*40), 30, I
280  PAINT  (50 + I* 50, I*40),  I, I
290  NEXT
300 LOCATE 21,5: PRINT "screen 1,";P2;
310 PRINT  " COLOR";CBK;",";PALET
320  LOCATE 22,5: PRINT "Use 'B' to change background";
330  LOCATE 23,5: PRINT "Use 'P' to cycle palettes";
340  LOCATE 24,5: PRINT "Press spacebar when done....";
350  RETURN
```

I use a variation of this routine in several of my games. It gives players the ability to configure the colors to their taste (or lack thereof). Let's look at how it's done:

First, we'll meet the new players:

|SCREEN| -- In earlier chapters we used SCREEN 0 to switch among text screens. Here, we use SCREEN 1,0 to tell Basic we wish to use graphics. This changes the orientation of the screen from 80 by 25 text characters to 320 by 200 pixels or dots. This lets us create graphics images, lines and patterns.

|COLOR| -- This command is similar to that used in text mode. However, the second argument can only be 0 or 1. The first argument still sets the background color.

```
|     COLOR 2, 0 |
```

This sets the background to green, and uses the 0 palette. CGA Graphics mode has two palettes -- 0 uses colors green/red/yellow-brown and 1 uses cyan/magenta/white. If you issue a palette change command, the screen stays the same and the color switch. An undocumented palette can be achieved by issuing the SCREEN 1,1 command. This is illustrated in the program. This palette contains cyan/red/white and can make your programs more appealing than the universal cyan/magenta of IBM graphics.

|CIRCLE| -- This command, not surprisingly, draws a circle of given color and radius.

```
|    CIRCLE (X,Y), radius, color  |
```

Here, the cursor will be at X,Y (in pixels), with 0,0 at the upper left hand corner. The radius is in pixels, and the color is 0 to 3.

|PAINT| -- fills an area with a color, starting at the indicated point. For circles or boxes, the center works well, but it's not required. The paint starts at that point and continues in all directions until a line of the indicated color is reached. Unfortunately, PAINT has a nasty habit of leaking if you try to PAINT an unclosed object, so use it with care.

The next program builds on what we've learned with screen and color and adds straight lines and patterns. (See program LINES.BAS, included in the shareware package.)

In addition to the commands we met earlier, this program introduces 2 new ones:

|PSET| (X,Y), color -- places a dot of color at the indicated pixel. You can think of this as anchoring the cursor, also, similar to the LOCATE command in text mode.

|LINE| -- This command has two forms, and several options. It's simplest, but longest form is

```
|      LINE (x1,y1) - (x2,y2),color    |
```

This draws a line from x1,y1 to x2,y2 in this color. From this point, we could write

```
|      LINE (x2,y2) - (x3,y3),color    |
```

to add a new segment starting at x2,y2, or we could just write

```
|      LINE - (x3,y3),color              |
```

which says to draw the line from the last cursor location to x3,y3. There's a simple way of drawing boxes and optionally filling them. Just add the commands B or BF to the LINE command:

```
|      LINE (x1,y1) - (x2,y2), c, b      |
|      LINE (x1,y1) - (x2,y2), c, bf     |
```

Here the two points represent the upper left and lower right corners of a box. The 4 lines represented by these corners are drawn automatically.

The exercises this time are a little more involved than previously. This is because Basic graphics commands are fun to play with, AND because there are several points that can best be made after you've had some time to experiment. Even if you don't do any of the exercises, you might find it interesting to read through this exercise section.

[... back to the Contents Table]

7.4 More Exercises

A. Create a program that randomly draws boxes and circles on the screen, filling them with color. Experiment with different colored borders and painting. Watch what happens when borders overlap.

At this point in the series, if you've been doing your homework, you should find the previous exercise straightforward. The next project should be more challenging!

B. This project is quite similar to the simple game proposed last time. The intent this time is slightly different. By using the function keys we can create a simple line drawing program.

1. First draw a large box on the screen. Don't allow the user to cross this boundary. (Solution hint: You'll need to calculate the x,y coordinates before you draw them & check them against some boundary conditions. For example, if you draw the boundary at the outer edge, using

```
|      LINE (0,0) - (319,189), 1, B    |
```

Then you need to check that any proposed x is between 0 and 319 and any y is between 0 and 189.

2. Use the program outlined last time (for interpreting the arrow keys) to create a new program that draws lines. The pseudocode will be:

```
{ draw a point in the middle of the screen}

if {left arrow pressed} then { draw line segment to left}

if {  up arrow pressed} then { draw line segment up     }

....
```

You can make the line segments some constant amount or a random amount.

2. Use F3 & F4 to increase and decrease speed. Remember to check for a speed of 0. Don't let the speed go below 0, though.

3. Use F5 to change the color of the lines being drawn

4. Use F6 to draw a circle at the current location.

[... back to the Contents Table]

7.5 For Super-Extra Credit

Last time we described a game in which you tried to control a moving cursor without hitting characters that were already on the screen. This time we didn't include those rules in the game description. Before reading further, try to think of reasons why this might not be as simple as it was with

characters. Can you think of methods by which you could get around these exercises? (No need to write the actual programs.)

Ready?

When you draw a line, how can you tell what pixels compose the line between the 2 points? One of the strengths of the LINE command is also a drawback. We just tell it to draw a line from (x,y) to (x',y'), but we don't have to calculate the individual pixels that get drawn. With characters, we know exactly which ones to look at. For the line, we'd need to get out our trig manuals to calculate exactly which pixels were being covered.

There are algorithms that describe how to tell if 2 lines intersect, or if a particular point is on a line, but even using these, we'd be stretching an interpreted language to try to do it all in realtime.

Don't feel too bad if you didn't solve this exercise. It's pretty difficult to find a solution that's both practical and fast. The point rather is that Basic provides an extremely flexible method for investigating such involved and difficult exercises. In several of my published games I've wrestled with these very problems. My approach is to write short Basic programs first, examining the difficulties in detail. When the problem's solved, I can then either compile it in PowerBasic or rewrite it in another language. So even if my final language is C or Pascal, Basic is often my first choice for quick & dirty graphics prototyping.

[... back to the Contents Table]

7.6 The End of the Beginning

This completes the BASIC TRAINING TUTORIAL. We've covered a large number of topics, and if you've done some of the exercises, projects and played with the example programs, you should have a solid grasp of the elements of Basic at this point. What you do next depends on the types of programs you want to write. If you're interested in studying more of what Basic can do, and what a compiler can add, the Advanced Tutorial will help. When you register, you get it for free.

ADVANCED BASIC. Topics include: Animation techniques, Shape Shifting techniques, Error Handling, Chaining, Windows, Peek / poke, Bload / bsave, Plus, details on differences between interpreters and compilers. And complete source for all examples. Advanced Basic is ONLY available to registered users.

Registration costs only $20, and you will receive The Advanced Basic Tutorial as a bonus, along with more source code for all the examples in that tutorial.

In addition, when you register you also get an evaluation copy of the LIBERTY Basic compiler for Windows. This program lets you develop Basic programs in the Windows environment, without the need for the Windows Software Development Kit.

You might also want to order the Games Package option. The Games package includes source code for 2 Cascoly programs that you can compile and modify for your personal use.

ATC -- An air traffic controller game that's perfect for a few minutes or hours of fun. Easy to learn, but difficult to master. The game tracks best scores at each of 20 levels of difficulty. Shows how to use real time interrupts and error detection.

ECOMASTER -- The CGA version of Cascoly's ecology game. A diverting ecology game in which you bid for and trade animals based on their abilities to thrive in different environments.

To Register, return to DOS, and enter the command:

```
| REGISTER |
```

An order form will be printed for you. Or send $20 + $4 shipping to:

```
| Cascoly Software   |
| 4528 36th Ave NE   |
| Seattle WA 98105   |
```

[... back to the Contents Table]

---=== T H E E N D ===---

HTML Conversion 2004 by Thomas Antoni www.QBasic.de