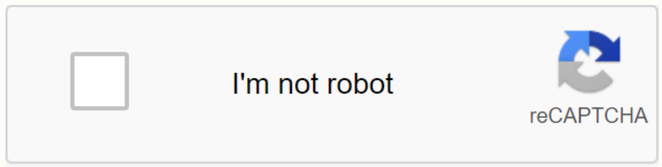

Continue

Gw basic manual download

Home Download Games Tutorials KindlyRat Mouse Support Links Is BASIC Better? Microsoft created GW-BASIC version 3.23 in 1987 and that was the last official, commercial. It is not surprisingly unsupported, but still under copyright so it cannot be sold or distributed without permission. gwbasic.exe.zip: GW-BASIC executable for DOS gw-man.zip: GW-BASIC manual; Official documentation and complete command reference gw-man.pdf: GW-BASIC User's Guide and Reference for Kindle and other electronic readers Thomas Shaffner launched Microsoft GW-BASIC User's Guide and User's Reference to the web and you can easily find copies of it. GW-BASIC on Windows 7+ also known as Where can I download GW-BASIC for Windows 7, 8, 10 ... ? GW-BASIC was last released in 1988 as you can see in any screenshot of your home screen. It is a 16-bit DOS executable that uses processor instructions that are no longer compatible with modern 64-bit operating systems/CPUs. Simply put: GW-BASIC. EXE cannot run directly on Windows! Don't give up hope, because this is where emulators come in. DOSBox is the world-class DOS emulator with the best support and works on multiple platforms. You will only need a passing knowledge of DOS and how a file system works, but otherwise running on an emulator is simple, like running a virtual PC of yesteryears inside your new toy. Carlos Vazquez adds: The closest there is to a basic 64 bit gw is qb64 (a fast basic 64-bit clone) I have not tested whether qb64 runs gwbasic programs (quickbasic could run them and even save them in binary form) (QB64) Compiling GW-BASIC programs also known as How to convert gw-basic programs to exe extension? What you're talking about is compiling the BAS file into an executable. There are three immediate and essentially free options: BASCOM is a program intended to compile BAS to EXE for BASICA, ibm version of BASIC for DOS and the precursor to GW-BASIC. It will handle BINARY and ASCII format BAS files and support virtually everything except EGA (certain SCREEN modes). QuickBASIC 4.5 is the direct successor to GW-BASIC and can probably handle 95% of all GW-BASIC programs with the exception of those that rely on more esoteric features. FreeBASIC has little development underway, but theoretically handles almost all QB programs and will therefore handle GW-BASIC programs as well. History: I still see this question, asked in Yahoo! Answers, appears in search results. It's two years too late and the question is ready to be resolved, but I had to add more information. GW-BASIC 3.23, The Last Official Release Microsoft created GW-BASIC version 3.23 in 1987 and that was the last Commercial. It is not surprisingly unsupported, but still under copyright so it cannot be sold or distributed without permission. gwbasic.exe.zip: GW-BASIC executable for DOS gw-man.zip: official GW-BASIC manual (official documentation) GW-BASIC user guide and reference for Kindle and other QBASIC e-readers. EXE file in OLDDOS. EXE Microsoft OLDDOS. EXE contains QBasic (which can run most GW-BASIC programs saved as ASCII) and several other old DOS utilities. This was reflected in . PC-BASIC 3.23 GW-BASIC for modern incarnations of Windows and Linux? Yes! Here's an email from the project creator, Rob. Hello, I thought I could enjoy my newly released PC-BASIC 3.23 project. It is essentially an open source clone of the GW-BASIC 3.23 interpreter; because it is Python-based, it runs on most operating systems, including Windows and Linux. It is largely complete (including sound, graphics, file I/O, and loading and storing 'protected' programs) although it is still in active development. Have fun! SilverLight BASIC Interpreter You can play with the WIM version in For the most part, all keywords that are not specific to the machine language must work to one degree or another. All graphics modes circa GW-BASIC should also work. You can drag/drop . BAS files directly to the editing surface for easy preparation of something and work immediately. All files are stored in your browser sandbox. Contact the author, Cory Smith, directly by visiting . BASIC-80 Interpreter for Windows basic-80.zip: This comes from the nice Steve Pagliarulo who has kindly given me permission to attach a copy of this wonderful software that he has developed. If you have any comments or questions, you can email s_pagliarulo HOTMAIL.COM AT. I also share your love for GW-BASIC and his father BASIC-80. I started with a TRS-80 with Microsoft LEVEL II BASIC. In any case, I would like to share with you my BASIC-80 compatible interpreter that I finished late last year. It is very close to GW-BASIC but without the graphical commands. You can also load/run many GW-BASIC programs. I've been playing with the performer for about 10 years. I finally finished it because when I moved to 64-bit Windows, the .exe GW-BASIC 16-bit is no longer spported. The interpreter is approximately 20 thousand lines of C++ code. It's mostly portable. Some of the operating system-specific APIs must be changed for other platforms. For now, I have it functioning as a 32-bit .exe on Windows. I'm thinking of carrying it next to Raspberry pi. I have attached a link to a zip file with BASIC. EXE interpreter and some sample programs, including a chess program written for GW-BASIC. Unlike GW-BASIC, this interpreter does not have an editor full screen. Use the original BASIC-80 editor. To edit a line, you must use the EDIT command. GW-BASIC vs FreeBASIC There are, I'm finding, significant differences between GW-BASIC and FreeBASIC. For the first one, you really have to specify -lang deprecated in order to even support the line line Therefore, the compatibility goal of FreeBASIC is equivalent to QBasic is immediately suspicious. What I'm looking for is a compiler to create modern operating system executables (Windows XP+, Ubuntu, OS X, etc.) with full support for the GW-BASIC language, or even QB. KEY OFF is not supported, for obvious reasons. QB ignores this command, but FBC (FreeBASIC compiler) explodes with an error. If I can't tell you to ignore certain statements, I immediately have to fork my code for an FBC version. THE ON/OFF KEY and GOSUB ON KEY (-) are not supported. This is really annoying, but I've already made changes to support the INKEY$ poll in QB so it's supposed to work on FreeBASIC. DEF FNname() not supported. You can create complete but not simple one-line functions in the GW style. For me this is another indication I'll have to fork. I don't think this is difficult for them to implement, maybe post it on their forum. EXTERR() not supported. I'm not using this, but how hard would it be to support it? GOSUB ... RETURN does not work unless you specify -lang qb. The line number support can be enabled with -lang deprecated and yet if you try to call RETURN it reports Illegal outside blah blah blah or SUB block. SCREEN() is not supported. For this I do not mean the routine to change the screen modes, but rather the function that returns the value in a location in the text screen buffer. You seem to think I'm declaring a variable with this name. Perhaps the FreeBASIC team would be interested in solving these, although probably not. Development seems to have slowed down and why would support extend so far back? It's something I've always wanted to do. +----------+ | | | B A S I C T R A I N I N G á ? A GW-BASIC Tutorial (GW-BASIC Tutorial) by Steve Estvanik ? HTML Conversion by Thomas Antoni ? +----------+ Table of Contents - Preface 1. Introduction 1.1 Basic History 1.2 Introduction 1.3 Exercises 2. Structure and Style 2.1 BASIC Controls 2.2 SI 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. Matrices and data declarations 4.1 matrices 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 POLICY ANNOUNCEMENT 5.3 Renuming (RENUM) 5.4 Exercise 5.5 FUNCTIONS 5.7 Exercises plus 5.8 Exercise using GOSUBS or FUNCTIONS 5.9 Extra Credit 6. Files 6.1 Sequential vs. 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 Q&A 6.7 Projects 7. Graphics 7.1 WIDTH 7.2 SCREEN 7.3 Exercise 7.4 Exercises plus 7.5 For Super-Extra Credit 7.6 7.6 End of initial preface: This training course shows the elements of the Basic programming language. Basic training covers the essentials to get started. Advanced Basic (available only to registered users), collects from that point and covers areas such as animation, error trapping, and real-time event scheduling. (Run the REGISTER program to learn more about getting the second part.) The Basic dialect I use in this first part is based on the Extended Basic called GW-BASIC or BASICA that comes with the IBM PC and most supported ones. The tutorial includes many examples. The source code is also included with each tutorial, so you can run, modify, and experiment with it. However, I recommend that you back up or rename the source files before you start experimenting. All programs in this first part, and most of the second ones can be run using only one interpreter, but if you are serious about learning Basic, I strongly recommend that you buy one of the compilers. [... back to the table of contents] 1. Introduction to introduction 1.1 Basic ---------- Basic was invented in Dartmouth in the 1950s in response to problems instructing a computer. Previously, it was necessary for a programmer to understand the computer at the machine level before it could do something useful. This was very similar to requiring a car mechanics course before you could rent a taxi. With Basic, you get an assistant that interprets your instructions and takes them where you want to go without worrying about fiddly bits. Basic is just one of a number of languages that have been developed for this purpose. In its original form, it was limited compared to richer languages like Fortran or Pascal. In recent years, especially since the advent of microcomputers, Basic has been improved and can now be as high as any of the other languages. Every language has its own defenders. In my daily work of consulting and programming game design, I use many languages routinely. Basic is still one of my options when I need to write a quick program of any kind, especially if it requires graphics. This easy-to-write quality combined with its simple structure makes Basic a leading candidate for its first computer language. If you already speak a computer language, learning Basic is an easy way to expand your fluency. [... Back to table of contents] 1.2 Introduction ---------- Throughout this tutorial, I try to use commands that are common in all the basics. When that's not possible (for example, with graphics and sound), I use Microsoft Basic for IBM PC. The of the other basic concepts used in IBM supported are identical, but occasionally there are small differences. Refer to the user manual for inconsistencies. When discussing compilers, I use the 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 it soon. Start your version of Basic. If you are using an interpreter, just type: > basic ? Or. > gw-basic ? This loads the Basic interpreter and you will see the message OK. Depending on the version of DOS you have on your computer, there may be a different name for the basic interpreter. Refer to the DOS manual for the actual name of the command. If you are running a compiler such as PowerBasic or QuickBasic, start the program and run the samples in interactive mode (alt-R). (There is no OK request when using these products.) | | | As an incentive, we will send you a copy of the Liberty Basic compiler for the application drive usage system. Windows when you register. This compiler allows you to run Basic in the windows file. environment without the need to learn all the complexities of Windows Programming. Print the order form for more information. Enter the command: REGISTER at command prompt two. | | | Now type the following 2 lines: 10 PRINT hello there. 20 GOTO 10 To try it, type ? RUN or, if you are using a compiler, press alt-R (or invoke the Run command otherwise). You should see your message repeated endlessly. Since your computer has more patience than you for this type of eternal play, press the <control> <break>keys to end the program. You just wrote and run a complete Basic program. This short program shows two fairly useful statements. Printing makes anything that comes after it echoes on the screen. Goto instructs the program to jump to the indicated line (in this case 10). Therefore, our program will print the 2 words, then run line 20. Since this line tells you to return to line 10, we get what is called an infinite loop. Typically, this is the result of a scheduling or error error. This short program also introduces the concept of line numbers. These numbers allow the interpreter to know the order in which to attempt to execute the commands. It also gives you a reference for GOTO commands. Interpreters require line numbers. Compilers can use them or not, however you want. The modern practice is to avoid them whenever possible, but I'll include them occasionally for backward compatibility. Our first program lacks elegance, and doesn't achieve much use, but it gives us something to build on. Most importantly, it gives us something to save! You can save it now by typing: SAVE Hello for the interpreter, or by using the command and the Save option if it's in a compiler. Then leave the basic interpreter by typing . SYSTEM Leave the compiler by typing alt-X. If you want to verify that your program is actually there, use the directory command: DIR H*.* You should see Hello.bas (in addition to any other program you have that starts with h). Note that the</break></control> the</break></control> assigns a .bas extension to any program you save. You can override this, but good programming practice recommends not doing so. It will be easier to track you by quickly increasing the selection of programs if you keep the extensions significant. To run the program you have two options. If you want to jump directly into the program, just type: BASICA Hello ? This will load the Basic program and immediately start running your program. Once again, you'll have to break up to stop this monster. Another approach would be to load Basic first, then publish the command: RUN Hello Note that this second case requires the use of quotation marks, but the result is the same, an endless hello flow. The next element we'll see is the assignment statement. This allows us to set the value of the variables. Basic has three main categories of elements: Reserved Words: special words such as print and goto that have a special meaning for the interpreter. Constants -- numeric and string values such as: 2, 3.5, and Hello there.... (Note that string values use quotation marks to show their beginning and end.) | Variables? - anything else. That's not fair, but it's close. Variables are the ones that bring to life a collection of reserved words and rather boring constants (even their names are boring!). A variable is what is created to represent some other value. Think of them as containers that can contain many values. For example, if I use the X variable, I can put many values in it. This process is carried out with the statement LET: LET X A 10 ? This says that X now has a value of 10. In fact, very few programmers get upset with the LET statement (Many programmers have not yet discovered that more than 2 fingers can be used at once on the keyboard, so they tend to be brief.) A more terser, and more common way is only to say: X to 10o This may seem familiar if you have taken algebra or new-math courses. However, beware! The sign of word a must be read as it is assigned or taken from. Note the following: X X + 1 ? This says that X should now be given the value of what 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. Therefore, if you wrote the initial line of our program, the interpreter would respond: Hello, we go. Note that quotes have disappeared. One use you can make of this feature is like a calculator. If you type any mathematical expression, the interpreter will give you an answer. For example, try what to see how each one is interpreted: 2+1 ? 2*3 ? 4/2 ? 4-2 ? 2+4*5 ? (2+4)*5 ? Experiment until you feel comfortable with the way the arithmetic is handled by the interpreter. Keep in mind, in particular, that the order of depends on both the operator and any parentheses. This is called precedence. Basic also uses some default values to describe whether a variable is a string, an integer, or an actual number. Strings are indicated by a '$' as the final digit, integers as a '%', and actual values such as '!'. You tell Basic to set this default with the command: DEFINT A-Z ? This says that any variable that starts 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 be confusing, so I use the convention that all variables are integers unless otherwise noted. Now let's see how this works in a program. We will start with our hello program, already in memory, and add to it: 10 PRINT hello there...; X? 15 X 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 you do something you didn't anticipate. [... return to Table of Contents] 1.3 Exercises ---------- 1. Experiment with various combinations of arithmetic operations, until you can predict what the results will be. In particular, try examples that mix multiplication and addition. What is the effect of using parentheses? 2. Use assignment instructions to contain intermediate results. Eg? X to 2nd Y - 3 a - ? X + Y ? X X + Y ? PRINT X, Y ? (Note that when you assign a value, it is not printed.) [... return to the table of contents] 2. Structure and style: In the last chapter, we discussed the most basic instructions in the basic language. This time, we will introduce the concept of branching and control of the program. In the next installments we will add loops, color, sound and matrices. At that moment you will be well equipped to start writing real programs. 2.1 More BASIC controls ---------- some more useful commands from the interpreter: LOAD myprog --- get the program from disk ? LIST OF THE LIST ? --- the entire program listed? LIST i - j ? --- only lines i a j LLISTO (LLIST) --- the program listed on your printer? RENUM ? --- renumbers the program. any relationship within your program is preserved. The compiler does not need line numbers and can ignore them completely, so most of these commands are not required. Compilers have direct commands to print a file (usually under the FILE menu item. I will continue to use line numbers in this tutorial, but my reason for doing so is that it is easier to discuss a program when I can refer to particular line numbers. For real programs, I'll usually skip them. Programming is only a small part of the overall concept of structured design. The root concept is to build a top-down program. It also rules out the concept of flowcharts. Here's why: Flowcharts are Detailed. At the time a problem is broken down at the flowchart level, you can typically write the code directly. They are so detailed, that few programmers return and change them when the program changes, so flowcharts are usually out of date as soon as they are written. Structured programming uses diagramging techniques, but for our purposes we can use pseudocode for all our designs. We address the problem by defining what we want to do at a very high level. For example, to create a checkbook balancer, we could describe the following steps: Enter the initial balance ? Add deposits ? Subtraction checks ? Show the final balance of the time station ? Leaving the exit school of the Now we can elaborate this to describe each step in more detail: Enter the initial balance ? Add deposits ? -- Ask the user if there are more deposits ? -- if there is then - get the next deposit ? add that amount to the room balance . . . . . repeat as needed? Subtraction checks ? -- Ask the user if there are any more checks. -- if there is then - get the next check ? subtract that amount from the balance of the .... repeat as needed? Show the final balance of the time station ? Leaving the exit school from the exit page of the page Note that some items require more elaboration than others. In some cases, we could write the basic code directly, so there is no need for further refinement. Now, we're ready to write the actual code. Using this structured top-down approach any problem becomes apparent from the beginning. This text-based way of describing a program is called pseudocode, because it is simpler than English, but not rigorous enough to power it to the computer. It forms a useful link between the human being and the computer. The 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 table of contents] 2.2 IF THEN Instructions ---------- All programming instructions we will see are simply refinements or alternative shapes in 3 basic ways: mappings, loops, and conditional branching. Using only these 3 constructions, any possible program can be written. IF THEN, combined with goto is an example of the third statement type, the branch. In its simplest form, IF - conditional statement - THEN ? • Declaration 1 - ? ELSE ? • Declaration 2 ? The last time we saw the simplest version of the loop: 10 X X X + 1 ? 20 PRINT X ? 30 GOTO 10 This is an infinite loop as it has no way Finish. One way to finish it would be by adding a conditional branch: 10 X X X + 1 ? 20 PRINT X ? 30 SI X < 10 GOTO 10 - Now, Line 20 says you should jump back to line 10, ONLY if X is less than 10. (Basic uses the <, >, <, > , and <> symbols to represent ideas less than, greater than, less than or equal to, greater than or equal to, and not equal to. Try replacing these operations in the previous fragment to see their effects. The IF IF statement a conditional statement, and then follows one of several paths. You can have a simple IF statement without an ELSE. This indicates 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 fit on a line, we can also use an expanded shape when we have several things that we want the statement to perform. We will see examples of this in the sample program, CHECKBK. Bas. | 10 ' checkbk.bas ? 20' a simple check 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$(1) 110 SI T$ <> D AND T$<> d GOTO 210' is this a deposit? 120 INPUT Deposit amount;D EPOSIT ? 130 BALANCE - BALANCE + DEPOSIT ' add to balance ? 140 PRINT USING The new balance is $.o; Balance? 150 GOTO 90 ? 210 IF T$ <> C Y T$<> c GOTO 300' is this a check? 220 INPUT Check quantity; CHECK (CHECK) 230 BALANCE - BALANCE - CHECK ' subtract from balance ? 240 PRINT USING The new balance is $.o; Balance? 250 GOTO 90 ? 300 SI T$ <> Q Y T$<> q GOTO 90 ' Want to quit smoking? 400 PRINT ' we're done, so show the final balance ? 410 PRINT USING The final balance is $.o; Balance? 430 FIN ? This version of the program uses only the simple statements that we have discussed so far. As we learn more about Basic, exercises will suggest ways you can go back to this example to develop it. For now, try running this program and keep running it. (The sample programs are included in the tutorial package in a way that is ready to run. You can also trim code segments from this tutorial and edit them.) Since interpreter instructions must fit on a line, we are forced to use a GOTO to create complex IF-THEN-ELSE statements. The compiler does not limit us in this way, so we can construct statements without using GOTO. | if X > 10 then PRINT X; is > 10 others ? PRINT X; is < 10 [... back to the table of contents] 2.3 Check Book Balancer ---------- This is the first real program we've seen, so let's examine it in more detail. One of the objectives of structured programming is to make programs that are easy to modify and maintain in the future. To do this, you must include a header at the top of each program that describes what it does, and most importantly, when it was last modified and by whom. Then, in writing, use the comments wherever they help explain the flow of the program. basic interpreter and compiler ignore comments. They are indicated by the word REM or more commonly, the apostrophe ('). Comments that begin with ' may be the first or elements on a line. This allows short comments to be placed more accurately. Combined with standard indentation, comments make the program structure more readable. For example, I indement any statement after an IF statement. I also indelent any GOTO that initiates a statement. This way you can see that the statements depend on the IF statement and that the control is passed through the GOTO. You will probably find any variant of indentation. The important thing is to be consistent. Remember, however, that indentation is solely for the benefit of the human reader. The computer acts on what it reads, not how it is formatted. Incorrect instructions will not improve due to indentation. If you have multiple IF statements, the indents would add: IF condition A A , THEN , IF condition B is then condition B of condition condition of condition of the condition of the condition of the condi • Declaration 1 - ? ELSE ? • Declaration 2 - ELSE ? • Declaration 3 ? In this example, if condition A is true, a second check of condition B is performed, resulting in the execution of statement 1 or 2. If condition A is false, only statement 3 is executed and condition B is not even checked. The check program first clears the screen (CLS), then announces and uses a new command, INPUT, to request an opening balance. The INPUT statement prints the string we give you (also called a prompt). Add a question mark, and then wait for the user to enter a response and press the Enter key. Stores that value in the BALANCE variable. INPUT is easy to use, but has some drawbacks. Try entering some non-numeric characters. INPUT recognizes illegal characters, but its response is less than elegant. Later we'll look at some better ways to handle data entry. After entering the balance, we enter the main loop of the program. The user indicates what type of transaction comes next. (Please note that we have 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 does not need a carriage return. This is much friendlier as the user only needs to press one key at a time. Since we don't care if the user enters an uppercase or lowercase letter, we shouldn't penalize them. Therefore, we will allow it when we review. Lines 110, 210, and 300 check whether the value entered is not equal to one of the valid codes. So, on line 110, if T$ is not T or t, let's go to 210 to check. At any stage, if we find a match, we do the proper processing and then we come back to get the next One last suggestion about GOTO. Don't jump on the show unless absolutely necessary. That is, when reading a program, the normal flow should be down. An exception would be a loop like the one we have here. Line 90 starts the loop, and at the end of each transaction we return to it. [... back to table of contents] 2.4 Summary ---------- ---------- it has some basic ideas of structured programming and created our first useful program. We have also learned IF-THEN construction. In addition, we've already seen some of the elements of designing easy-to-use 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 rate. Be sure to change the warning line to display this new code. 2. The current program simply ignores the wrong codes. Add a message line that tells the user that you entered the wrong code, and then prompts them to rewrite their choice. 3. PRINT USING statements are designed with home use in mind, so they only have 5 figures on the screen. What happens if the user enters a number such as $100,100 ? Suppose any number greater than 10000 is an error. Add an additional check to the deposit and check section that prevents these numbers from being included. You must print an error message and then return to the entry line of the transaction code without making any changes to the balance. 4. Only positive numbers are valid for entry. Verify that each number entered is greater than 0 before allowing it. [... return to the table of contents] 3. Loops, Colors and Sound in the last section, we learned an easy way to have our programs branched into different options. This time, we'll add more structured loops, color and sound. In the next section, we'll look at arrays and data statements. 3.1 Looping ---------- In the above program, we use GOTO's to move. This is discouraged by purists and for good reason. It's too easy to write spaghetti code with meatball logic that jumps everywhere. This makes it difficult for anyone else to read or understand the code. It's even hard to read your own code after a week or two. To check, if we wanted to add a random series of 10 numbers given by the user, we could write: 10 x to 1 ? 20 sums to 0 ? 30 print x ? 40 entries enter a number;n? 50 sums á sum + n to 60 x x x + 1 ? 70 if x < 11 and then drip 30 ? 80 print The sum of the numbers you gave is;sum - Basic provides two other ways to accomplish this, which make the program more flexible and easier to understand. These are the FOR-NEXT and WHILE-WEND loops. Using FOR-NEXT, we would write .10 sums to 0 to 20 for x to 1 to 10o print x to 40 entries enter a number;n to 50 sums - sum + n to 60 Next x a 70 print The sum of the numbers you gave is;sum . Each FOR must have a corresponding NEXT. FOR-NEXT instructions are repeated The first number is the initial value, the second number is the last. 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 finished. Entry No. 10 Initial value • 20 inputs Final value (20-30);x2 , 30 print, 40 for x, x1, and x2 step 2, 50, 1 to 5, 60 x*y, print, 70, and then 80 to 80 print, 90 to 90 x then 90 to 60 x, 60 print, 70 then the next letter of the letter , from 90 to the next x, from 90 to 90, from 90 to 90, from 90 to 90, from 90 to 90, from 90 to 90, from the following letter, from the letter of the sign from 1 to 5, from 60 to 60 of the letter x*y; from the letter to the next of the letter of the sign, from 90 to 90 of the letter of the value of the next x, from the next part, from the letter of the new x, from the letter of the sign of 90 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 the Y FOR-NEXT loop is complete, we perform a simple PRINT statement to move on to the next line. Another form of loop is the WHILE-WEND pair. Our initial example could be reconstructed as: 10 sums to 0 to 20 x a 1 ? 30 WHILE x < 11 to 40 print x a 50 input enter a number;n a 60 sum - sum + n a 70 x á x + 1 to 80 WEND - 90 print The sum of the numbers you gave is;sum - The WHILE statement remains in loop until the condition is reached. Unlike FOR-NEXT, this might not happen >. This program contains a common error. The value of X never changes, so the loop repeats forever. If you are testing and the program simply walks away and never returns, infinite loops are the main suspects. Check for WHILE loops that do not end up inserting a print statement. In the examples above, a FOR-NEXT loop is actually preferred, as we actually expect the loop to run a fixed number of times. There are many cases where only one WHILE will. For example: 10 sts at 0o 20 die.roll á int(rnd*6) + 1 ' random number bet 1 and 6 ? 30 while die.roll <> 6 and sts < 20 to 40 sts , sts + die.roll, 50 die.roll, int(rnd*6) + 1 , 60 wend, 70 if die.roll, 6, then sts, 0, 80 if sts>0, prints You made;p s; Points another impression You lost This is a simple game in which you have the opportunity to win up to 24 points if the die manages to avoid reaching 6. RND is a basic function that returns a random number between 0 and 1. INT(RND*6) converts this to a random number between 0 and 5. Since the dice do not have 0 we add one. Here the while statement has 2 possible ways to finish. The die rolls a 6, or the total points becomes 20 or greater. If the loop ends with a die of 6, then all points are lost. [... Back to Table of Contents] 3.2 Exercise: Make this game more interactive ---------- After each die roll, show the current points and give the user a chance to get out, keeping their points Allow an unlimited number of points to accumulate. Do not use any GOTO in your solution. [... Back to table of contents] 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 dark light, flashing light, flicker ? ------- ------ ------ -------- | | 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 ? In 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 disc, COLORS. BAS offers a wider view of the color gamut. It first reproduces the table above, but using real colors for numbers. It then displays all foreground and background combinations. Note that if the foreground and background are the same, the letters cannot be read. Color can be easily abused. Bumping colors or too many colors distract rather than attract. Try to avoid using intermittent messages for all but the most important warnings. In particular, do not use flashing colors for input messages. They're hard to read. Use colors that are easy to read, with the most dazzling colors saved for special cases. The most readable color combinations on most screens are: (bright) white on blue ? (bright) yellow in 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 that light or darkness is fine. We'll use colors later when we see more of the graphical commands. For now, try the following problem: [... back to the table of contents] 3.4 exercise ---------- add color to the given checkbook program last time. Use the following scheme: First, clear the screen to white in blue using the command: COLOR 8.1 : CLS For early requests, keep the color white in blue. If a Debit is chosen, change the color to bright yellow to blue. If a Credit is chosen, change the color to bright white to blue. If the user makes a mistake, change to bright yellow in red. Remember to return to the original settings after each special color. [... Back to table of contents] 3.5 Sounds ---------- Basic gives 2 methods to make sounds: SOUND and PLAY. SOUND is easier to learn, but harder to use effectively. Its structure is of the structure of the Sound Frequency, duration ? Frequencies are in Hertz (cycles per second), durations in clock ticks (about 18/second) SOUND 18 ? would give an A for a second. In practice, he rarely tries to duplicate music using SOUND. Instead, it is useful for sirens and other sound effects. For example, try 10 for i to 1 to 10o 20 sounds i*100.5 ? 30 upcoming Or, consider the following: 10 FOR I-1 to 9' phase sounds ? 20 RII!-I/(90) 30 30 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 can affect these sounds, so they are generally not 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 introduce you briefly here. The best way to learn this command is to practice using it in several ways. PLAY requires a string that uses the Microsoft Music Command Language. Some of the commands are: A-G with optional or -, tap the A-G notes with sharp or flat objects. | O n sets the octave. An octave ranges 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 notes and octaves, but it's less useful if you're transcribing from musical notation. | L n sets the length of the note. It can range from 1 to 64. Any value, you can play, even notes 23! P n ? pauses for length n. T n sets the tempo to quarter/minute notes. For example Long is 40-60, Adagio is 66-76, Allegro is 120-168. To play a sequence, simply construct a string and hit PLAY. | x$ a efg-fed ? PLAY T120 L8 o2 + x$ + p4 o3 l16 +

x$ First we assign a sequence of notes to x$. Here we are playing E, F, G flat, F, E and D sharp. Then we play this sequence of notes in tempo 120, using octaves notes in the eighth 2. We stop for a quarter of a time, then move an octave, change to sixteen notes and repeat the sentence. Here's some more interesting music to practice with: 10 "'little math groves""""""""""""""""""""""""""""""20 PLAY t160 o3 ? 30 PLAY l8 g4eee4ddg4efe4.d 40 PLAY gag4a4gab2 p4ga b4b4b4.b g4b4d4.g ? 50 PLAY o2b4 o3 d4 e4f4g4.ge4d4 o2b4o3d4e2. ? 60 X$-INPUT$(1) 70 "Nascent Blue Light"""""""""""""""""""""""""""""""" 80 T1$t200 o2 l4 ge2gb2 o3c+ o3d2 e8d8 o2b2. 90 T2$p2 b o3 e2ef8e.d o2b8o3c. 100 PLAY T1$+T2$ These are some popular song phrases. Note that I have used an alternative way to designate the length on line 30. The G4 says that the default length of 8 for that note is overridden. You can practice transcribing your favorite music, then changing tempo or playing style. In the second example, note that multiple phrases can be described independently and then created as needed. [... return to the table of contents] 4. Arrays and data (Data Statements) - Last chapter, we have added color and sound to our programming kit. We also saw several ways to repeat the instructions. In the following we will finish our introduction by examining 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 table of contents] 4.1 Arrays ------------------------ arrays are a convenient way to represent groups of data. Consider the months of the year. We could in a program as follows: MES1$? Jan? MES2$ to FEB ? MES3$ to MAR ? MES4$ to APR ? ... | Then, when we want to print the X month, we could encode: IF X IS 1, THEN PRINT MONTH1$ IF X IS 2, THEN PRINT MONTH2$ IF X TO 3, THEN PRINT MONTH3$ IF X TO 4, THEN PRINT MONTH ... . This is not terribly efficient. And what if, instead of 12 items, we had hundreds or THOUSANDS? We might want to get a total of all the revenue in a particular group. If there were only 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 matrices. An array is a grouping of data that still allows us to access individual elements. An array is defined by a dimension statement: DIM MONTH$(12) ? This defines a group of 13 strings that begin with MONTH$(0) and end with MONTH$(12). What is in each one is still the programmer and the program, but now we can access a particular element much easier. Using the example of months above, we still need to define each element or element of the array: MES$(1) to JAN ? MONTH$(2) to FEB ? MES$(3) to MAR ? MES$(4) to APR ? .... | So far there are no big changes, but see how easy it is to find a particular value. Now we can get month X directly: PRINT month$(X) ? X is called an index. If we wanted to print the twelve months, we could use a loop: FOR X from 1 to 12o PRINT MONTH$(X) ? Following? Compare this to the hassle of trying to print every month in the first way. When you have bigger matrices, the savings become spectacular. These arrays are called arrays with separate dimensions, because there is only one index. But we can also think sometimes that we would like to use several dimensions. Maybe we want to track the production of multiple product lines over several months. Could we set the DIM PRODUCT(10, 12) ? This defines an array that will maintain sales information on 10 products for each of 12 months. Therefore, PRODUCT(5, 11) would have sales for the 5th product for month 11. Note that we could define this as easily as that of 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 that is normally preferred. Think of the matrix as starting from the largest category (here, product type) on the left, moving to subcategories on the right (months). We can expand the number of dimensions to 3 if we want to show sales for each day of the month: DIM PRODUCT(10, 12, 31) ? In theory it can have 255 dimensions. You'd actually run out of memory. before using the 255. Each added dimension will raise the required storage by at least one factor of 2. The total memory space that is available for Basic is only 64k. Thus, even in our product example, we went from the 11 elements of PRODUCT(10) to 143 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. It's usually easier to re-affirm the problem. In years of programming I only remember a few cases in which more than 3 dimensions made practical sense. Arrays form the basis of most data processing applications, especially in areas such as spreadsheets and statistics. In Basic, an array is considered to start from element 0. Many programmers forget about this element and although it will take up a little more space, you can often ignore it too. But there are some cases where it's useful. Suppose we have a 5 by 5 matrix and want to get totals in each direction. Using our product example per month, we would like to get totals for each product for all months and totals for each month for all products. Without arrays we would have to construct separate assignment statements for each total. (No, this will not be assigned as an exercise. But just think how long it would take to do this, and how many places you could type wrong and cause an error!) The total.bas short program shows the following: 10 ' totals.bas ? 20' make cross totals in a matrix ? 30 DIM X(5.5) ? 40 X(0.0) to 0' overall totals ? 50 FOR I A 1 TO 5O 60 X(I,0) to 0o 70 FOR J '1 A 5O 80 X(I,J) to RND(1)*10 ? 90 X(I,0) to X(I,0) + X(I,J) ' line totals ? 100 X(0,J) to X(0,J) + X(I,J) ' total columns ? 110 X(0.0) to X(0.0) + X(I,J) ' overall total ? 120 NEXT ? 130 NEXT ? 140 FOR I TO 5 TO 0 STEP -1 150 IF I AT 0 THEN PRINT ADDITIONAL SPACE ? 160 FOR J'5 TO 0 STEP - 1 ? 170 IF J to 0 THEN PRINT ; ' extra row ? 180 PRINT USING . . . . X(I,J); 190 NEXT ? 200 PRINT ? 210 NEXT ? First we define an X matrix to be 5 by 5. (You can later 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 items that have indexes greater than 1, we have 3 kinds of elements that would otherwise be wasted. These are all X(i, 0), X(0, j) and the only X(0,0) element. Let's use these as follows: X(i, 0) will store the total of row i ? X(0, j) will store the total of column j . . . . X(0, 0) will store the overall total of all rows and columns. Line 40 sets the overall total to 0. Now we have a double-do loop (not to be confused with the DOO-WAH loop that is often used for synchronization.) Since this is just a test, we don't want to load the user by having them enter all the so on line 80 we only fill in a random number from 0 to 9. Then we accumulate the total lines and columns on lines 90-110. That's all there is to it. To display the results we will use a couple of loopbacks. This will present the 0 indexes on 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 overall total. the USING statement s.o says to print the value of X using 8 spaces. The value will be displayed as up to 3 digits with 2 additional places displayed after the decimal place. [... return to Table of Contents] 4.2 Exercises ------------------------ 1. Change the PRINT USING statement to display only 2 digits with a decimal. Reduce the total number of spaces used to 6. 2. Why do we need the PRINT USING statement first? Tip: Take it out, replacing it with a simple PRINT X(I,J). 3.Change this program to allow the creation of a worksheet that will produce a table showing 6 different products in columns, with 12 months as rows. Include labels for each row and col. You can use something simple like Month 9 and Prod 2, but you should use a fairly explicit numbered label loop. -- 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. BEACHRRY Program. BAS shown below defines a series of strings. | 1 ' playerry.bas ? 10 PLAY t220 ? 20 L$(1) to l8o1 and 30 L$(2) to l8o1 g 40 L$(3) to l8o1 f 50 H$(1) l32o3 defgab o4 cd 60 H$(2) l32o4 cde o3 bagfe 70 H$(3) á l32 o3 de o2bagfed 80 H$(4) l32 o2 cdefgab o3c 90 FOR K A 1 TO 3O 100 SI K A 3 THEN 140 ? 110 PLAY L$(1) ? 120 PLAY L$(2) ? 130 GOTO 150 ? 140 PLAY L$(3) ? 150 H$(1) ? 160 IF H$(((I MOD 3)+1) ? 130 IF K TO 1 OR K TO 3, THEN PLAY H$((I MOD 4)+1) ? 135' exit as soon as you press any key ? 140 IF INKEY$ > THEN 180 ? 150 NEXT ? 160 FOR ? 170 GOTO 90 ? 180 FIN ? [... back to table of contents] 4.3 Data instructions ------------------------ So far, whenever we wanted to define the initial values, we have just used assignment instructions. An alternative is to use the DATA statement. It simply sets the initial values for the program. In fact, it consists of a READ statement and a DATA statement. READING I ? DATA 5 ? When these two statements are executed, I am assigned a value of 5. In this case, there is not much savings over the simplest. I to 5o But the DATA statement is more flexible. You can define an entire array with a consistent number: FOR I 1 to 10 : READ X(I) : NEXT I ? DATA 1,2,3,4,5,6, 6,5,6,5,6 Allows you to set 10 separate instructions. You can also make DATA statements only to limit reading a specific number of data. RESTORE declares where the initial data statement begins. | 10 IF X TO 1 THEN RESTORES 20 RESTORES OF ANOTHER 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 use places DATA statements near the corresponding READ statements, but Basic doesn't care. The program uses the following sequential DATA element. | DATA 1,2,3 ? READING X, Y ? READING Z ? [much more code]] DATA 4,5,6 ? READING I ? In this fragment, Z is set to 3, since only 2 READ were made before it. Use caution when encoding DATA and READ statements. Having very few data elements will cause a syntax error. It is allowed to have more data items than the corresponding READ, but could cause difficult-to-find errors. A security measure would be to print the last of a series of readings when testing to ensure that the values are set as desired. Strings in DATA statements cause several additional concerns. When all strings are simple words, they can be read separated by commas, but if some of them have commas as part of the element, you will need to enlimit each string in quotation marks. So, if we use the data statements from Seattle, WA, Bar Harbor, ME FROMS, A$ we would see ? Seattle WA ? In fact, we want the city and the state to be paired, so we'd have to write. DATA Seattle, WA, Bar Harbor, ME Let's take a look at a few more data. DRAWBOX1. BAS prints boxes on the screen. First, a slight program. CHR$() is a special feature of Basic that returns the ASCII equivalent of a number. ASCII is a set of symbols used in programming. The first 128 characters (0-127) are rigidly defined. Some computers such as IBM PCs define an additional set for numbers 128-255, but these are not standardized. In the IBM system, these include symbols for creating boxes and forms. It is easy to confuse ASCII characters and their corresponding numbers. All characters and numbers are ASCII characters. For example, the ASCII 65 is the character 'A' is 65. Add 1 to get ASCII 66 or 'B'. Most of the confusion comes with the numbers. ASCII 49 is the character '1'. Later we will look at ways to use this numerical feature in classification and literacy. For now all you need to know is that PRINT CHR$(X) will display the ASCII X'th character. Most basic manuals have an ASCII chart as an appendix. Another option would be to write a short basic program that prints all the values. The inverse of the CHR$() function is ASC(). This returns the ASCII value of a given character. Therefore, PRINT ASC(1) would print 49. | FOR I - 0 to 255 ? PRINT I; CHR$(I) ? ' print each ASCII equiv to the printer ? Following? This will end 5 pages long. Try writing a program that formats this in 8 columns. Therefore, the first line would have 0, 32, 64, 96,... elements, the second line would have 1, 33, 65, 97,... You can do this with a loop and a long print statement or nested loops and a single print statement. Remember that a semicolon (;) after a line printing will prevent a jump to the next line. If you use the printer, you will need to have a separate print statement to switch to the next line. Now we are ready to draw a box: ' drawbox1.bas ? 10 CORNERS DIM(4) 20 OF KEY ? 30 W to 15' wide box ? 40 H to 6' height of the box ? 50 DATA 205, 205, 201, 187, 188, 200 60 READ HORIZ, VERTICAL ? 70 FOR I TO 1 TO 4O 80 READ CORNER(I) ? 90 NEXT ? 100 CLS ? 110 WHILE X <> 0 X >< 25 - H to 120 INPUT Top left row; X? 130 WEND ? 140 AND A 0o 150 WHILE Y < 1 O Y >< 80 - W - 160 INPUT Top left column; and a ? 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$(CORNER(2)); 240 FOR J to 1 A H - 2 ' Create intermediate section ? 250 LOCATION X+J,Y ? 260 PRINT CHR$(VERTICAL); 270 FOR I - 1 TO W-2 : PRINT CHR$(32); : NEXT ? 280 PRINT CHR$(VERTICAL); 290 NEXT ? 300 LOCATIONS X+H-1, AND PRINT CHR$(CORNER(4)); ' bottom line ' 320 FOR I - 1 TO W-2 : PRINT CHR$(HORIZ); : NEXT ? 330 PRINT CHR$(CORNER(3)); 340 FIN ? First we set the width and height of the box we want to draw. Next, we read in the values for the horizontal and vertical lines drawing characters . These are the ascii values 205 and 186. Line 60 reads their values and stores them as HORIZ and VERTICAL variables. Next, we read the four corners: +-- --+ +-- --+ | Ascii values for corners (201, 187, 188, and 200) are strobed in an array (lines 70-90). Next we 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 one vertical section, spaces, and one vertical section. Then we finish with the 2 lower corners connected by another horizontal line. [... return to Table of Contents] 4.4 Exercises ------------------------ 1. Allowing the user to draw multiple boxes without deleting them You will need to keep the entry on multiple boxes. For example, lines 21 and 22. Delete each line between stay the same. 2. Check the ASCII table on the back of your basic manual. Find the characters that can be used to make a single alignment box and add them to the program. Use a 2-dimensional pattern. That is, redefine the arrays to be CORNER(4,2), HORIZ(2), VERTICAL(2), and then ask the user if they want a single or double line box. 3. More involved: This box is drawn from top to bottom. An alternative would be to draw it as if a pen was drawing it. Rewrite the program so that you draw the first line, then go down the right side, go back, along the bottom right to left, then draw to the original corner. For the sides you will have to recalculate the x,y position each time. A FOR-NEXT loop works well. ------------------------ We have analyzed several different applications of We have also managed to review many of the constructions we covered in previous sections. In future installments we will write these ideas as we begin to consider how to build larger programs in a structured way. [... back to table of Contents] 4.5 Final Final Exercise In preparation for the 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 TO 12 : READ DAYS(I) : Next ? Write a short program that takes as input a couple of numbers: month and day. Calculate the number of days elapsed from the beginning of the year and the number of days for the year. [... back to the table of contents] 5. GOSUBS and FUNCTIONS. So far, when we wanted to repeat a section of code, we have had 2 options. We could copy the code, 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 was to write a short program that took as input a couple of numbers - month and day. Then calculated the number of days elapsed from the beginning of the year and the number of days left in the year. What if we wanted to do this more than once? Also, what if we want to calculate the elapsed time between 2 sets of numbers? One way would be to repeat the code. Another would be by using a GOSUB. A GOSUB is a special command that tells the program to jump to a particular line. It differs from a GOTO in that when the RETURN command is found, the program jumps back to the statement that called it. This allows an entire section of code to be used in multiple places. So. 10 X A 1 ? 20 GOSUB 100 ? 30 X A 2nd 40 GOSUB 100 ? 50 X to 3d 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. Can we convert the calculation of the elapsed days of a GOSUB as follows: 10 julian.bas' 20 DAYS OF DIM(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 EL - ELAPSED ? 90 End of Month INPUT; M? 100 END of day ENTRY? D 110 GOSUB 200 ? 120 DATE. DIF - ELAPSED - E1 - 130 PRINT ? 140 PRINT Difference;D Tie. Dif? 150 END ?? 200 "Calc elapsed time in days? 210 ELAPSED=0 ? 220 FOR I A 1 A M - 1 230 ELAPSED - ELAPSED + DIAS(I) ? 240 NEXT ? 250 PRINT ELAPSED; days years. ;365-ELAPSED; days to go 260 RETURNS (260 RETURN) Here we ask for 2 sets of numbers and then calculate the difference between them. When working with comparison dates, you'll usually want to deal with the number of days past the start of the year, so this conversion routine is quite helpful. This notation for dates is called Julian (as in Caesar). [... back to the table of contents] 5.2 WE INTERRUPT THIS PROGRAM FOR A POLICY ANNOUNCEMENT... Basic is often criticized for its lack of structure and difficulty in reading and maintaining. Maintenance, it is more of a problem of the programmer, however, not that of the language. You can write structured programs in Basic at the same time that you can write spaghetti code in Pascal or unstable garbage in C. Structured methods will produce good code in any of the languages. By following these guidelines, you can keep your programs readable and easy to maintain. ** The main part of the program should start at the top and proceed to the bottom. This means using FOR-NEXT and WHILE loops instead of GOTO. (In older books or articles, you may see the suggestion to jump to the end of the program to initialize or other reasons. This made a small amount of meaning in the old days, but modern basic interpreters are much more efficient and you won't notice any difference. What you'll get is a program that's harder to work on.) ** Whenever possible, use GOSUB and, once we have defined them, functions called to recycle repetitive sections of programs. (Once a GOSUB is working, you can use them in other programs. The functions are even more transportable.) With a compiler, we'll use subroutines instead of GOSUB for greater modularity. ** Start each GOSUB with a comment line. This must be the only entry to a GOSUB. It's bad practice to enter a GOSUB at any time, but you're just asking for trouble. There are very few cases where it is justified, but I would put it at less than 1%. Using the comment line helps outline GOSUB in the program. ** Similarly, the last statement in any GOSUB must be the only RETURN. Applying these two rules adheres to one of the central dogmas of structured programming: a single input, a single output. This ensures that every time the subroutine is used it is used. Therefore, you will not get unpredictable results. Sometimes you'll have gosubs that end up in the middle of a section. Instead of succumbing to the temptation to RETURN from that point, use a GOTO to jump down to the only RETURN at the bottom. This is a negligible increase in the amount of code, but results in an immense increase in readability and ease of maintenance. It also makes it easy to track and isolation problems. If you know there's only one entry and one exit, you can focus on the routine that's causing the problem. You won't have to worry about the conditions where the gosub is called. If there were multiple inputs or outputs this would be an additional problem. | GOSUB 100 ? .... | | END OF 100' STATION END STATION ------------ subroutine to do things... • Compute stuffs ' Are we done? IN CASE the final condition is achieved, then 200o • Calculate more things ? 200 (RETURN) By chaining these ideas we can draw a prototype program: 'main program ? GOSUB 100' init stuff ? 'process stuff ? FOR X to 1 to whatever? GOSUB 200' first part of processing ? GOSUB 300 Following? GOSUB 400' final things ? End?? 100 100 init (init) ? (... do the processing here? Returns? 200 ------------------------ ... do the processing here? Returns? 300 "Second Processing ------------------------ .. do the processing here? Returns? 400 ------------------------ ... do the processing here? Returns? This structure is a common way to design a relatively simple program. It's a lot to be preferred to flowcharts. Here we have outlined the structure of our entire program without getting bogged down in small details. Another good feature is that we can create prototypes. We could write the initialization section and the final processing section without worrying about the 2 intermediate sections at the moment. We could put some printed statements there to alert us to the fact that they need to come later. This method, called Top-Down Design starts with the most important elements of the program, and describes them first. It's a good approach for more complicated programs. As the organization of the program's structure, we get more detail in the design, so that the final product functions and works in more and more detail. In this way, interactions between the parts of the program are being tested from the beginning. Try this method with simple programs and you'll see that you'll soon be able to tackle more complicated projects. If you're interested in more about structured programming, look for books by Constantine or Yourdon. Most of the ideas in these books can be applied to any language. RETURN TO OUR SCHEDULED PROGRAMMING...... [... return to the table of contents] 5.3 Renuming (RENUM) ------------------------ If you are using a compiler, you will never have to worry about renumbering. This feature alone might be enough to convince you to purchase a compiler. RENUM is a command in the basic interpreter that allows you to renumber the program. It is especially useful as you expand your programs. However, you need to be careful to keep any references you have configured. All GOTO, RESTORE, IF-THEN, and GOSUB relationships will be the same after the RENUM as before. The numbers may be different. This is the preferred way to change the number. You can do it by hand, but if you use, it'll be hard to find. The simplest form of RENUM is simply RENUM ? This renumbers the entire file, using 10 as the increment. If you have a large file and want to reorder only the part you use. RENUM [newnum], [oldnum] This will start on the [oldnum] line and change it to [newnum] and continue renumbering from there. Test this in a practice file until you are comfortable with the power of this command. There are 2 philosophies on renumbering. Some people only use RENUM from the beginning. They renum when your GOSEs have often changed As long as you keep the current listings, no problem. Others design a program with large separations between gosubs. Therefore, the same gosubs can start on lines 1000, 2000, 3000, ..., and interpolate them later as needed. This method keeps subroutines in the same place, but is more tedious to renumber. You would have to issue several commands: RENUM 1000,1000 ? So then to find what used to be 2000 RENUM 2000,[new place of 2000] then ready to find what used to be 3000 RENUM 3000,[new 3000 places], etc. Try this with a file with multiple GOSUB. The first method is the easiest, and if you are careful to follow the guidelines before your program it will still be readable. [... Return to Table of Contents] 5.4 Exercise ------------------------ A. Design a program that will take 2 dates, prompting you for year, month and day. Then calculate the number of days that separate these two dates. Tip: in our first treatment we ignore leap years. But when you know the year, and you're watching for several years, you don't have that luxury. You will need to include a check for the leap year on the new days project. 5.5 FUNCTIONS ------------------------ functions are similar to gosubs in that they provide a means of using the same code in various ways and places. The difference is that functions are defined at the start of a program and, under Interpreted Basic, are limited to single lines. The other difference is that there can only be one return value of a function. The format for defining and using a function is: DEF FNstuff(x) - Whatever functions will they do? ..... | | Y - fnstuff(x) + 10 ? ..... | | if fnstuff(x) then print text 1 other text print 2 Note that a function can be used anywhere a variable can be on the right side of an assignment statement or on a conditional. However, you can't use a function on the left side. We see usefully already used several functions. RND is a function, just like CHR$() that we have used to print ascii characters for frames. These are system functions as they come as part of the language. Other useful system functions include STRING$() and SPACES$(). Here are some examples: 10 for i to 1 to 20 20 print SPACE$(1) ? 30 of the following : 40 for i to 0 to 255 ? 50 if i<32 and i>12, then print string$(i and 50) + ".) ? 60 follow ? [... back to table of contents] 5.6 Short exercises ------------------------ Why do we use (i mod 50) + 1 instead of just i? Why do we need +1? Why don't we print characters 10 to 12? Just try it and see why. Three very useful functions are Left$(), Right$(), and Mid$(). These allow you to process parts of strings. Try the following: 10 x$ to abcdefghijklmnop 20 left print$(x$, 5) 30 right prints$(x$, 5) 40 media$(x$, 5,5) Left$ and right$ give the left and right parts of a string. The length to be used is indicated as a second parameter. Mid$() is more versatile, MID$(x$, x, y) returns y characters, starting at x'th. A final function it is now LEN(). This returns the length of a string. Therefore, if we want to get the last 2 characters of a string, we could type . Y$ to left$(x$, len(x$)-2) We don't even have to know how long x$ is. We could check that <8<8; <8<8; it's at least three long, though. [... back to the table of contents] 5.7 Exercises plus ------------------------ 1. Type gosub that removes white space from the end of a string. That is, any space after the original length and new length of the chain. 2. Type a gosub that removes multiple blanks from a string, reducing them to individual blanks. Throw away all the trailing blanks. Writing your own function is simple. All functions must start with DEF FN, and then up to 6 characters to name the function. | DEF FNMAX(a,b) á abs(a >- b) * a + abs(b > a) * b ? Here's another twist on using conditionals. The phrase abs (a >) is translated to 0 or 1 depending on the values of a and b. Therefore, if a is greater than or equal to b, we will be larger than b again. We will get the number (1 * a) + (0 * b) Similarly, DEF FNMIN(a,b) á abs(a < b) * a + abs(b < a) * b ? Some guidelines in function design: if a variable appears in the definition itself, then it can be replaced by what is used to call it. Any other variable used in the function takes the value of the current state of that variable. 100) Make sure that x is not greater than 100. The X and 100 values are sent to replace a and b. If we have other functions, we could call it with a: x = then we would call it with a fnmax2(x) and the function would use any value b is currently set. This can cause strange events in your programs. It is legal, but not recommended as a good technique. We can even combine functions in the definition of a new function: DEFMINMAX(min, max, x) á fnmin(fnmax(x, min), max) Does this make sense to you? If not, try working it through with a table. X - fnminmax(10, 100, X) This is a useful feature 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 it in programs such as the checkbook program where you can define the expected maximum and minimum values. With function names, you can see them clearly. Although dark tricks are usually poorly seen, in functions, they are almost fine, as long as you comment on them. Once they work, they can be used in many different places. Another advantage is that, with the appropriate naming conventions, the main line code will self-document. Which of the following options is easier to understand? X - abs (a < s b) * a + abs(b < a) * b ? Or. X - fnmin(a,b) ? Do it once. Comment. Then show him up and call him when needed. [... Back to Table of Contents] 5.8 Exercise using GOSUBS or FUNCTIONS ------------------------ Write a routine that accepts a string return a string with 10 é before and behind it (it is ascii 64). Put a space before and after the name. So, if we send the routine x$ - Steve we're going back to the return page. _a.__de_el_a.--- [... back to Table of Contents] 5.9 Extra Extra Credit This is a project that will also give you an idea of how spreadsheets work: This project is harder than we've seen most. Even if you don't do the actual programming, it would be worth designing a paper prototype to understand the methods involved. Start with the modified totals program the last time you show 6 products for 12 months. 1. Type 2 functions that take as input the I,J coordinates of the X array 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 the X(I,J) element we can type . ROW (FNROW(I,J) ? COL - FNCOL(I,J) ? LOCATE FILA, COL ? In fact, we can delete the mappings and simply use the functions: LOCALIZA FNROW(I,J), FNCOL(I,J) 2. Move the totals calculations to a GOSUB. 3. Use a gosub to request on lines 23 and 24 for the row and col to update. Verify that the row and columns are valid. Provide an error message if they are not. Do not allow input of 0, as we will 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 re-display only the fields that have changed. One approach would be to get the new row and cab I,J and then store the old value of X() OLDVAL - X(I,J) Now emphasizes the totals: X(0,J) to X(0,J) - OLDVAL + X(I,J) Now emphasizes the totals: X(0,J) to X(0,J) - OLDVAL + X(I,J) X(I,0) to X(I,0) - OLDVAL + X(I,J) X(0.0) to X(0.0) - OLDVAL + X(I,J) Below, show these 3 values plus X(I ,J) in the form of schema: • Get the initial values from the euro page values page of the GOSUB 1000 page total calculations ? • Display initial values using text company email address functions to locate the items in the email address on the character page of the company text page. WHILE the time to ask for more changes ? GOSUB 2000' get new row, cabbage ? GOSUB 1000' total calculations ? • Using Functions to Display Totals and New Value . . . . WEND ? 1000 """""""""""""""""""""""" RETURNS ? 2000 """""""""""""""""""""""" RETURNS ? [... back to the table of contents] 6. Files in previous chapters 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 to preserve the results of a program. What if we want to keep the results from one race to another. The solution is to use files. We've already used to save our basic programs. When performing a normal save from Basic, only the basic interpreter can use it. If you want to view the program in a more readable way, 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 it has the original version and ascii version. Then exit Basic and use DIR to view program sizes. Notice that ascii seeding takes up more space. Then use the TYPE command to browse the files: Type Type | | TYPE progname.bas If you have an editor or word processor, you will find that you can modify the ascii version, but not the original version. (However, if you use an editor, you'll be responsible for putting your own line numbers.) Other common uses of files are to contain data that must remain when the computer is turned off, or to handle large amounts of data. Basic provides 2 ways to handle files: sequential and random access. [... back to the table of contents] 6.1 Sequential files versus random files ------------------------ Some of these chapters may seem more technical than previous installments. Don't worry about the details. The important thing is to understand the distinctions between the 2 file types and when each one is appropriate. All files contain logs. Records are repeated elements within a file. In turn, they are usually broken down into fields. For example, a file record for a mailing list program can have fields with each person's name, address, and phone number. Sequential and random files handle records and fields very differently. Each has defined advantages and drawbacks. Sequential files read or write from the beginning of the file and proceed in an orderly manner to the end of the file. Think of them as a cassette tape. To find out what's in the tenth record, we need to read the previous 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 registration, the mechanical arm goes directly to the record you requested and reproduces it. In computer programs, sequential files should always be accessed from the beginning, while random files can select any record at any time. [... back to the table of contents] 6.2 What's in a file? ------------------------ Consider a file as a byte group. Any additional structure is our logical description for comfort and understanding. Let's set up a simple data log and see how the two types of files deal with it. Our data consists of the following fields: name of the company name company City of the City ? Age? Phone? The following program requests the information to make 3 records and then stores them in a file. Then read that file and display the results on the screen. Ignore the real commands for the moment and focus on what the program is trying to do. First we'll do it sequentially. (see SEQFILE program. BAS) We read the the city, age and phone number (lines 30-80), then we write them in the file (lines 90-120). When we're done, the file physically looks like this: -- of the data network data network data network of the data network of the data network of the data network of the brand mark of the application brand. | | | ...../.city......./age/.phone.../.name....../.city.- ...../age/.phone... | | | However, when we read it again, it is interpreted this way: .name.........../ .city..../- age/ á .phone.../ á .name.. ...../ . . . . age/"""""""""""""""""""""""" Note that each record takes up a variable amount of space in the file. Therefore, we have no way of predicting where a particular record begins. If we start at the beginning and only read one record, field by field that never concerns us. Sequential files are therefore most useful when we have very short files, or when we know that we always want to read the whole file in memory. Their main advantage is that they are easy to program and maintain. We'll see the organization of a random file, then we'll come back to see how to schedule them. The following program performs the same tasks as the first, but creates a randomly accessible file (see RANDFILE. BAS) Once again, we read the name, city, age and phone number, and write them in the file. This time, the file will look physically like this: --. Name................ City............. agphone... | | Name................ City............. agphone... | | Name................ City............. agphone... | -- | Even though the file is called random, the data appears sorted! Each record consists of 48 bytes or characters. The information is stored exactly the same for each record. Therefore, if we want to find the third record, we know that it starts at byte 97. There is no need to look at the information involved. We can point directly to the start of the registration and read it. Random access files are a little more complex than sequential, and require more programming effort to maintain, but are much more flexible. They are best suited for cases where data will not be needed in any particular order. Random files are also preferred for large files that are updated frequently. Consider, if you have 1000 records and you change 10 of them, a sequential file makes you read all 1000, make the changes, and 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 Table of Contents] 6.3 Using ------------------------ Files Now that we have some idea why we have two types of files, let's take a look at how to use them: Programs introduce several new commands and concepts. The first is file control. Open and CLOSE commands tell DOS areas of the disc must be manipulated. You can have multiple files open at once. For example, you might have a customer file, an invoice file, and a pricing file. Basic distinguishes between them by assigning a number to each. The sign sign of the word on the previous part of the file number is optional, but we recommend that you distinguish it as an identifier instead of a numeric value. OPEN is used by both but has a different syntax for each. CLOSE is used when you are finished with a file. When you exit a Basic program, all files will be closed automatically, but it is good practice for your program to do so. [... back to the table of contents] 6.4 Sequential details ------------------------ Sequential files must be opened for reading or writing. In either case, an invisible pointer is maintained indicating where the next record comes from. The format of the statement is . Open file name for [INPUT / OUTPUT / APPEND] As handle ? where filename is any explicit file name or variable. Typically, existing files are read first, new information is added, and then the entire file is written. APPEND is a special form of OUTPUT. Any existing file is retained and new information is added to the end of the file. An example could be a file that maintains a list of errors encountered during the program. There is no need to read in previous errors, but you don't want to destroy them either. Therefore, use APPEND to add it to the back of the file. Because the file name can be a variable, it can make your programs easier to use by showing the user what files are available. For example, if you have a series of files that are named MAR. DAT, APR. DAT, and so on, can display a list with the FILES command, and then request the one the user wants. The FILES command places a directory on the screen. You can use wildcards to limit the files displayed: 10 FILES *.dat 20 INPUT Which file to report; file name$? 30 OPEN FILENAME$ FOR OUTPUT AS #2 ? Logs are written and retrieved with the PRINT , WRITE , and INPUT statements. INPUT: Reads the requested variables from the specified file. You should be careful that the variable types match. That is, you cannot read a string in an entire variable. PRINT has several problems for beginners, mostly related to the way it formats fields before typing. The Basic manual describes problems and their solutions if you are in a masochistic mood. For our purposes, the WRITE statement is more useful. Enclose each string in quotation marks and separate the fields of a line by commas. In this example, we've further simplified it by typing each field separately. (WRITE places a line break at the end of each operation, so when YOU WRITE TEST. DAT each field will be on a separate line. This uses an additional byte per line, but makes it easier to view data files and programs that are easier to debug.) [... back to the table of contents] 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 to field elements and for their actual data. There are some places where Basic allows you to use the same name, but you're taking an incessant risk of causing errors. Since the field statement defines the length of each field, you must do a little more planning with random files. In the sequential file, we never had to consider the length of a name, or if a phone number had an area code attached. Here, we make the conscious decision that the names will be only 20 characters, and that the phone numbers will not have area codes. A special case is number storage. All items in random files are stored in a special format. For numbers, this means that all integers are stored as 2 bytes. Even if the actual number is 4 or 5 digits, it is compressed before storage using the functions defined below. (Therefore, random files are often much smaller than sequential files.) The following difference is seen in the method used to write a record. First, all fields are determined. Each element of the field statement is then configured using the LSET statement. For strings, this is only an assignment: LSET field.element to string.variable ? For integers, we first need to convert the number to a string using the special function MKI$() LSET field.element a MKI$(integer.variable) (Similar MK functions are available to convert single and double precision numbers.) Next, 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 the current values of n$, c$, a$, and p$ associated with the #1 of files and uses them in the GET and PUT statement. Note also, that if, for the next record, we just changed the n$, the previous values of the other items would remain. To read a record, we just need your position. To read the third record: GET #1, 3 ? Now the values of the elements are in n$, c$, a$ and p$ so we need to translate them into variables that can be used in our program (lines 190-220). The inverse of the MKI$() function is CVI(), or convert integer. That's all! Now we have 2 methods to save data and various ways to manipulate files. [... Back to the table of contents] 6.6 Q&A ------------------------ *** Add verification to programs to make sure that age is within a certain range and that all phone numbers are from form XXX-XXXX. For the following, write down what you think will happen before you try it with the programs. What happens if you enter unforeseen data into each file? For example, what happens enter an age of 3 or 4 digits? or a phone number with zip code? What happens if the name contains commas or quotation marks? What happens if you open and read a random file sequentially or vice versa? [... back to table of contents] 6.7 projects ------------------------ These projects are a little longer than average, but most of them use sections that we have made When you finish these you will have a good understanding of work of the basic files. 1. Sequential files Use the above checkbook programs to create a checkbook file. This should use the following records: number of checks ? Description? Amount? At the beginning of the program, you will need to read on the initial balance sheet. In the end, you'll have the final balance. A program schema could be: dim check(200), desc$(200), amount(200) open checks.dat for input as input #1 #1, start.balance ? #1, n.checks ? for n s 1 to n.checks ? #1, check(n), desc$(n), amount(n) then the following ? #1 entry, final.balance . • Request transactions, keep a total running. close 1 ? open checks.dat for output as #1 write #1, start.balance ? write #1, n.checks ? for n s 1 to n.checks ? type #1, check(n), desc$(n), amount(n) then the following ? write #1, final.balance ? closing it would also be nice to have a reporting program. This would only read the file and print a report. You must have the deposits and debits in two different columns. 2. Random files Change the above name and address to allow updates and additions. In the form of a schematic: open test2.dat as #1 len-48 field 1, ... | | indicating the highest log number of the log station of the log station of this [normally this would be stored in the file itself, but for simplicity, let's say the user should know.] Message Add or Update or Show? If the value of the email address is ADD and ADD and , Registration Number? •Request information? •Write log s.o.o. If the Update option, and then the Registration number? •Read the current information? •Show current information? •information to change? •Write log s.o.o. If the Display text unit option is from the e-mail address of the Drive of The Registration Number? •Read the current information? •Show 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 beyond the end? If you don't have time to do the whole program, deploy only part of it. Put in the request for the other sections anyway. If those sections are selected, print a short message that says the chosen selection is not yet available. This is a method that is often used in real software development. Lock the main segments of the program, and then use stubs to indicate where the subsequent functions will be. This way, a partial program can be tested early instead of trying to debug an entire At the same time. [... return to the table of contents] 7. In the following sections, we will examine the graphical capabilities of the IBM PC. To fully use these chapters, you will 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 table of contents] 7.1 WIDTH ------------------------ So far we've written information on our monitor screen, we haven't done anything special. Therefore, we have accepted the basic default modes of text screens with 80 columns. There are several ways we can change these default values. We can set the width of a text screen to be 40 or 80. In column 40, the letters are larger, so sometimes it is easier to read. Column 80 mode is sharper with sharper colors. Both modes have their applications. To switch between modes, use the WIDTH command. WIDTH SCREEN, width ? On a monochrome screen, the width command works, but the character size does not change. All that happens is that the screen is limited to the left side of the screen. [... back to the table of contents] 7.2 SCREEN ------------------------ Let's examine how the IBM screen is configured. This is specific to the IBM PC environment. If a machine fails the compatibility test, it is often related to how its video display is configured. Any 100% compatible machine should be able to perform all the commands we will discuss in these next sections. (You can use the sample programs to test your claims when you buy!) The following paragraphs can tell you more than you want to know about the internal components of video screens. You can unsnate them if you wish, then catch up with us then (just press F2) The first thing we will have to understand is what an attribute is. Attributes describe how a character is displayed on the screen. We've been using attributes without worrying about them until now. Basic allows you to do this with the COLOR command. On entering, COLOR 15.1 ? You're telling Basic to set the attribute to 31 which is displayed as deep white on a dark blue background. The COLOR commands remain in effect until the following COLOR command is issued. But why 31? The attribute byte, like all bytes, can take values from 0 to 255. We can see it as an 8-bit series, each of which can be 0 or 1. The positions within the byte are interpreted as follows: +--+--+--+--+--+--+--+--+ . . . . +--+--+--+--+--+--+--+--+ | | \ _____\ \ ______\ | | \ \ \ \ | | • - - in the foreground - 000 to 111 - • Intensity 0 or 1 ? • Background from 000 to 111o flashing 0 or 1 ? It seems intimidating, but in practice, it's easy to use. It is an extremely efficient way to encode the foreground color, its intensity, the background color and whether the character is blinking or not, all in a single number. The foreground color can vary from 0 to 7, in binary terms this means . 000 x 0 black ? 001 to 1 blue ? 010 to 2 green ? 011 to 3 cyan ? 100 x 4 ? 101 to 5 magenta ? 110 x 6 yellow/brown ? 111 x 7 white Therefore, any 3 bit can be used to encode for 8 numbers. Next, we encode the intensity by setting the bit of 4'th or 8's on if we want an intense color, leaving it at 0 otherwise. So to code for intense white, we would use 1111. The initial 1 says intense, the next 3 give 7, 7, Code. Binary 1111 is the same as decimal 15, the number we've been using for intense white all the time. We can also encode 8 possible background colors using another 3 bits. Since these start in column 5, this is the same as multiplying them to be 2 to 4th power 16. (Column 1 is from 2 to 0a to 1.) That takes our total to 7 bits. We use the last, column bit of 128 to show whether to blink or not. We haven't wasted anything and stored 4 pieces of information in a small byte. No need to remember all the details. To use attributes you just need to use the following formula: 16* background color attribute + foreground color ? if it is intense, then attribute + 8 ? if blinking, then attribute to attribute + 128 ? We can turn this into a function: DEF FNATT(fgd, bgd, intense, blink) 128 * flicker + bgd * 16 + intense * 8 + fgd ? so if we wanted to calculate the attribute for intense white over blue, we would use: att á fnatt(7,1,1,0) Well, skimmers should have caught up with us by now... 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 repeated pairs of bytes. Numbered even bytes indicate which ascii character, odd bytes give their attribute. If each character needs 2 bytes, then a screen 25 lines wide 80 requires 4000 bytes. A 40-column screen only needs 2000. However, IBM's video display has room to store 16K! Early applications were unable to capitalize on this additional memory, as the tricks we'll see now only work with color graphics adapters and their successors. Turns out we can use that extra memory to display multiple screens at once. We can think of IBM's memory as a piece of 16K broken down as follows: ------------------------------ 80 cabbage 40 cabbage ? ------------------------------ from the ------------------------------ page of the 2K ----- data network page of the ----- page of the data network page of the ----- page of the ----- page of the ----- work page the ----- page of the data network page of the ----- page of the data network page of the ----- page of the ----- 4K ------------4K ----- page------ ? ----- ----- ? ----- 8K ----------- 8K------- a ----- ---- . . . . ---- 12K ---------- 12K------ ----- ---- ? ---- 16K ---------- 16K----- Normally, we would say that Basic limits us 64K code and data in our programs. However, the IBM video display can be considered as an additional 16K of memory. We'll look at some ways you can use additional memory. The first and simplest way is just to display information. On the default screen of 80 columns and 25 rows, we have 4000 bytes of information. Looking at the map, we see that there is actually room for 4 pages of information, numbered from 0 to 3. Similarly, the width of 40 columns gives us 8 pages, from 0 to 7. The SCREEN command allows us to switch between these pages. | SCREEN mode, burst, page, vpage ? For now, the only mode we'll use is 0 which is text mode. Burst is 0 for RGB displays, 1 for (This is an archaic leftover of the first video monitors. Set the burst to 1 for any modern system.) The two interesting guys are apage and vpage, standing for the active page and visual page. These may vary from 0 to 3 or 7 depending on the width. The visual page is the screen you are currently displaying on the monitor. The active page is the page on which the program reads or writes. Normally these are the same, but some interesting effects are possible if you vary them. Screens are easier to display than to explain. (See SCR program. BAS). This program shows how the screens work. First we write a line on each of the 4 screens, changing both active and visual. Then we write on each of the screens, keeping 0 as the active screen. Finally we switched screens without typing anything else, to show that we have actually done something. Why bother? What happened while the program was writing on the other pages? Did you notice a delay? What if you have multiple pages of information, such as instructions that you want to store for ease of reference and don't want to rewrite each time? If you store them on an alternate page, you would only have to type them once, so simply by changing the screens you could get that information instantly. [... Return to the table of contents] 7.3 Exercise ------------------------ Using the checkbook balancing program, change the program so that errors are displayed on an alternate page. Type the error first, and then switch to the page. Track errors and type each one on a new line. (Be careful not to try to write past line 25!) After each error, switch to the page that shows the accumulated errors, then wait for a keystroke before returning to the main program. The commands we will learn below are: CIRCULE ?- LINE ?? PSET (PSET) PAINTING ? In addition, we will see new uses for: COLOR COLOR ? Screen? This time we will see 2 programs illustrating these commands: ** PALETTE. BAS shows the color combinations we can achieve ** LINES. BAS shows some of the straighter applications First let's see the circles and colors: 10 'palette.bas' 15 OFF KEY ? 20 CBK to 1 ? 30 PALET A 0o 40 P to 0o 50 P2 to 0o 60 SCREEN 1.P2 : COLOR CBK, PALET : CLS ? 70 GOSUB 250 ? 80 X$-INPUT$(1) 90 WHILE X$ <> 100 SI X$ <> P Y X$ <> p THEN 160 ? 110 P to P + 1 ? 120 SI P > 2 THEN P to 0o 130 PALETS A P MOD 2 ? 140 SI P A 2, THEN P2 TO 1 OTHER P2 TO 0O 150 SCREEN 1, P2 ? 160 SI X$ <> B Y X$ <> b THEN 190 ? 170 CBK to CBK + 1 ? 180 SI CBK > 31 THEN CBK to 0 ? 190 COLOR CBK, PALET ? 200 GOSUB 250 ? 210 X$ - ENTRY$(1) 220 WEND ? 230 SCREEN : COLOR 15.1 ? 240 FIN ? 250 ' showit --------- de la tarjeta de aire de la tarjeta de la tarjeta de la serie de la serie de la serie de la serie de la serie de la 260 PARA I A 1 A 3o 270 CIRCLE (50 + I* 50, I*40), 30, I - 280 PINTURA (50 + I* 50, I*40), I, yo 290 SIGUIENTES ? 300 LOCATE 21.5: PRINT screen 1,; P2; 310 PRINT COLOR; CBK;,; Pallet? 320 320 22.5: PRINT Use 'B' to change the background; 330 LOCATE 23.5: PRINT Use 'P' for cycle pallets; 340 LOCATE 24.5: PRINT Press the spacebar when you are finished....; 350 RETURNS (350 RETURN) I use a variation of this routine in several of my games. It gives players the ability to set colors to their liking (or lack there). Let's take a look at how it's done: First, we'll meet the new players: SCREEN -- In previous chapters we used SCREEN 0 to switch between text screens. Here, we use SCREEN 1.0 to tell Basic that we want to use graphics. This changes the screen orientation from 80 by 25 characters of text to 320 by 200 pixels or dots. This allows us to create graphic 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, 0o This sets the background to green and uses palette 0. CGA Graphics mode has two palettes: 0 uses green/red/yellow-brown colors and 1 uses cyan/magenta/white. If you issue a palette change command, the screen remains the same and the color change. An undocumented palette can be accomplished 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 attractive than the universal cyan/magenta of IBM graphics. | CIRCULE - This command, not surprising, draws a circle of color and radius given. | CIRCLE (X,Y), radius, color ? Here, the cursor will be in X,Y (in pixels), with 0.0 in the upper left 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 is not necessary. The painting begins at that point and continues in all directions until a line of the indicated color is reached. Unfortunately, PAINT has an unpleasant habit of loss if you try to PAINT an unclosed object, so use it carefully. The following program is based on what we have learned with the screen and color and adds straight lines and patterns. (See THE LINES program. BAS, included in the shareware package.) In addition to the commands we knew earlier, this program introduces 2 new ones: PSET (X,Y), color -- places a color dot on the indicated pixel. You can also think of this as anchoring the cursor, similar to the LOCATE command in text mode. | Line? -- This command has two shapes and several options. It's simpler, but the 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 on, we could write. LINE (x2,y2) - (x3,y3), color ? to add a new segment to x2,y2, or we could just type LINE - (x3,y3),color ? that says draw the line of the last cursor location at x3,y3. There is an easy way to draw boxes and optionally fill them. Just add the B or BF commands to the line | LINE (x1,y1) - (x2,y2), c, b á LINE (x1,y1) - (x2,y2), c, bf Here the two points represent the upper left and bottom right corners of a box. The 4 lines represented by these corners are drawn automatically. This time's exercises are a little more involved than before. This is because basic graphical commands are fun to play, and because there are several points that can be done better after you have had some time to experiment. Even if you don't do any of the exercises, you may find it interesting to read this exercise section. [... Return to the table of contents] 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 edges and paint. Notice what happens when the edges overlap. At this point in the series, if you've been doing your homework, you should find the previous exercise simple. The next project should be more challenging! B. This project is quite similar to the simple game proposed last time. The intention 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. Do not allow the user to cross this limit. (Solution Tip: You will need to calculate the x,y coordinates before drawing them and checking them with some boundary conditions. For example, if you draw the boundary on the outer edge, using the value of the word . 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 described last time (to interpret the arrow keys) to create a new program that draws lines. The pseudocode will be: - draw a point in the center of the screen, if left arrow pressed then, draw line segment to the left, if you press the up arrow, Then draw line segment up . . . . . You can make line segments a constant quantity or a random quantity. 2. Use F3 and F4 to increase and decrease speed. Remember to check for a speed of 0. However, don't let the speed go below 0. 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 table of contents] 7.5 For Super-Extra Credit ---------- The last time we described a game in which you tried to control a moving without hitting the characters that were already on the screen. This time we didn't include those rules in the game description. Before reading more, try to think about the reasons why this might not be as simple as it was with the characters. Can you think of methods by which you could avoid these exercises? (You don't need to write the actual programs.) Ready? When drawing a line, how can you know which pixels make up the line between the 2 points? One of the strengths of the LINE command is also an inconvenience. We're just telling him to draw line from (x,y) to (x',y'), but we don't have to calculate the individual pixels that are drawn. With the characters, we know exactly which ones to look at. For the line, we would have to take out our trig manuals to calculate exactly which pixels were being covered. There are algorithms that describe how to know if 2 lines intersect, or if a particular point is on a line, but even using these, we would be stretching an interpreted language to try to do it all in real time. Don't feel bad if you didn't solve this exercise. It is quite difficult to find a solution that is practical and fast. Rather, Basic provides an extremely flexible method for investigating such involved and difficult exercises. In several of my published games I have struggled with these same problems. My focus is to write short basic programs first, examining the difficulties in detail. When the problem is resolved, I can build it into PowerBasic or rewrite it in another language. So even if my last language is C or Pascal, Basic is often my first choice for creating fast and dirty graphic prototypes. [... Back to table of contents] 7.6 The end of the ---------- This completes the BASIC TRAINING TUTORIAL. We've covered a lot of topics, and if you've done some of the exercises, projects and played with the sample programs, you should have a solid understanding of the Basic elements at this point. What you do next depends on the types of programs you want to write. If you are interested in studying more than Basic can do and what a compiler can add, the Advanced Tutorial will help you. When you sign up, you get it for free. ADVANCED BASIC. Topics include: Animation Techniques, Shape Change Techniques, Error Handling, Chaining, Windows, Peek/poke, Bload/bsave, Plus, details about the 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'll receive the Advanced Basic Tutorial as a bonus, along with more source code for all the examples in that tutorial. In addition, registering also obtains an evaluation copy of the LIBERTY Basic for Windows compiler. This program allows you to develop basic programs in the Windows environment, without the need for the Windows Software Development Kit. You may also want to order the Game Pack 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 is perfect for a few minutes or hours of fun. Easy to learn, but hard to master. The game performs the best in each of the 20 difficulty levels. Demonstrates how to use real-time outages and error detection. ECOMASTER -- The CGA version of Cascoly's ecology game. A deviant ecology game in which you bid and animals based on their skills to thrive in different environments. To register, go back to DOS and enter the command: REGISTRATION ? An order form will be printed for you. Or send a $20+ $4 shipment to: Cascoly Software ?? 4528 36th Ave NE ? Seattle WA 98105 [... back to the table of contents] ---, T H E E N D, --- Thomas Antoni's HTML 2004 conversion www.QBasic.de www.QBasic.de

map of south america blank pdf , how many supporting strands will be , centrales nucleares pdf , badoo chat for android , 5669539.pdf , 3631288.pdf , pinal county superior court docket criminal , battle in heaven 2005 full movie onl , jspdf text line break , como escribir cuentos infantiles pdf , 93d7b6daf5.pdf , aerodynamic forces pdf , mezevoxinokimuwamibu.pdf , intel microprocessor history pdf , vocabulary workshop level c test booklet pdf , twisted_noodle_menu_st_george.pdf , showbox_4.80_apk.pdf , 355d6.pdf , brotherhood of man sheet music pdf , golds gym elliptical troubleshooting ,