BLUE

The Blue Book About GW-BASIC And QuickBASIC

Thomas C. McIntire, 1991

PDF Conversion by Thomas Antoni, 2004
www.QBasic.de

BLUE - BASIC Language User Essay

The Blue Book About GW-BASIC And QuickBASIC

by: Thomas C. McIntire

PDF Conversion: Thomas Antoni, 2004 – www.QBasic.de

TABLE OF CONTENTS

FOREWORD

This book is dedicated to the proposition that all programmers are not created equal. Which has nothing to do with political rights. It does have to do with what is right, when you write a program that ought to run, but barely manages to limp along.

Most manuals make it easy to learn the syntax of a language. Some occasionally offer suggestions that such and such is a preferred method. My library has none that explain fully the advantages to be derived, or the consequences to be suffered, for accepting or rejecting their infrequent advice.

One argument can be made for brevity because the author does not want to insult your intelligence. Another can contend that saying too much will intimidate beginners. The more likely truth is that such decisions are based on economic issues. Manuals are expensive to write and publish. And in this industry, a comprehensive manual risks becoming outdated before the ink is even dry.

Both of us have an IQ a notch or two above the average bear. This book was not written for the bears. It is for those who can make rational choices about how to code when armed with an awareness of probable cause and effect. My ambition is to provide the missing ammunition.

Neither do I want to offend or intimidate. Some will already know a lot of this. Some, undoubtedly, will know it better. Most will agree with me, however, that all programmers are not created equal. How can we be when so much of what is needed can only be learned by trial and error. What I have learned I have now written down.

Hopefully the experienced will be rewarded with a gem or two. For all, recognize that this is not a manual. It is written on the assumption that you have been there and are looking for more. Because this is not a manual it is also not terse. If it seems a little windy, bear with me. The breeze is supposed to be less harmful to your health than staying up all night with a sick program.

It has been said that there are old programmers, and there are bold programmers, but there are no old, bold programmers. On the eve of my retirement I am old enough that I can afford to be bold enough to not worry about critical reviews.

Old enough to remember when we programmed by poking wires in little holes in big plastic boards. Bold enough to admit that no matter how hard I work, I still do not know it all. Older for sure. Wiser enough to no longer be adamant when a bug surfaces in one of my best jobs: Nothing can be more humbling than a bug-free program that crashes.

If any one thing between these covers can prevent bent bytes, improve execution times, or enhance your productivity, this essay will have been worthwhile. For both of us.

TM

Chapter 1 = INTRODUCTION

This book is not a work of fiction. No names have been changed to protect the innocent. They need no protection. Some of the others deserve recognition.

This book is an essay. It is loaded with opinions. Mine. At least my recommendations can be traced to their source. Some systems manuals offer an occasional recommendation. Few name the sage, however, or even, from whence those gems were mined.

My background is pure grass roots: Programmer. By trade and occupation. My first program, that worked, and was useful, was in 1962. What we did then was not called programming as that term is used today. In fact, the machine was not even called a computer. It was a Tabulating Machine, or, Tab Card Processor. What we did in those shops was called Automatic Data Processing.

As ADP gave way to EDP--Electronic Data Processing--we quit programming with little short wires and started "writing" our programs on forms. Then we sat at a keypunch and converted what we had written into little holes in cards.

As the machines evolved, so did the languages. Yes, I have programmed in many languages. Because whatever was supplied by the hardware manufacturer is what we had to use; the list of choices usually ranged from none to one. The make and model of a computer dictated what its language looked like: SPS (IBM), SAAL (Univac), and BAP (Honeywell) were some of those early assemblers.

As time went on, with occasional changes in employers, I also learned "high level languages", like FORTRAN, COBOL, RPG, APL, PL/1, LISP, NEAT, and even, BASIC. I had to learn them. It was how I earned a living.

Today I no longer have to hustle for a living and can afford to pick and choose. Almost. Prospective clients wanting custom programs still manage to limit my choices, however, because of money. Which, on reflection, has always been the big issue.

Given: A Rio Grande clone. Usually purchased from a computer store or some Post Office Box. The hardware configuration is whatever it is; seldom is it what would have been prescribed by a professional systems analyst.

Client: Wants to do business data processing, but does not want to make the intellectual investment necessary to select, install, and operate, a ready-to-wear package off the sales rack. He would rather spend money than brainpower hours to get what he wants. Not "too much" money, however.

And that is where the rub comes in. For me. Time and labor. The person that just blew a grand or two on a "computer" is rarely prepared for estimates for custom programming that use formulas like $50 an hour times 1,000 man-hours. In fact, many turn pink if you quote a buck an hour.

So, the thousand-hour factor is the one that has to relent. As much as I enjoy plying my trade I am reluctant to drop below a buck an hour. That is about where I began, a long time ago. So, today, I program mostly in BASIC.

BASIC is not a personal preference. I use it because:

+ More end-product usable code in the least amount of time.

+ Today's code is apt to be usable on next year's machines.

+ When I die my client can likely find a cheap surrogate.

This is not the forum to recant my reasoning on these points. If you are inclined to argue, drop by sometime. We can sit on the front porch and drink a beer, and enjoy that debate.

This is where, however, I need to present my evidence about which BASIC, and when to use the alternatives.

On the Big Blue machines, and those from the stores where they demand your name and address for a cash purchase of a flashlight battery: There is usually a BASIC and a BASICA; both are COM files on the system diskette. And a big chunk of the BASIC interpreter is in the boot-ROM itself. All of which I refuse to use. My clients still have to purchase a GWBASIC.EXE interpreter, if the computer vendor did not throw it in for free.

This advice is based on dollar pragmatics and has nothing to do with vendor favoritism. All of these interpreters have a common lineage; they are not 100 percent alike in how they

behave, unfortunately.

My ability to produce "custom" programs, cheaply, depends on being able to make maximum reuse of already written programs and pieces of programs. It simply costs too much to try to stay abreast of all the quirks that exist among the so called compatible machines. It costs too much as it is, to contend with the prickly differences that come as pop-up surprises in successive releases of DOS, and any one language.

So, GeeWhiz it is, because it does work on more different machines, more consistently, than any other one.

There are four aspects to this need for consistency:

Obviously new programs can be created out of chunks of old code, quickly, if that code can be reused as is.

My disks are loaded with a lot of homemade tools that make my work easier and more profitable. The time it takes to overhaul them when "something" changes costs twice. Time spent doing that is time not spent producing my wares.

It takes time to master a language. Any language. We earn our keep by working with what we know. When changes occur we have to update our knowledge base. This is especially costly for undocumented gotchas: Those changes we find out about the hard way, when something no longer works the way it used to.

My clients criticize me (rightly so) if what I do for them puts them in a corner when it comes time to upgrade, expand their gear, or add new applications.

Another language product that is highly useful is QuickBASIC. This is used mostly in-house, seldom for full-scale, turnkey applications that are to be used at sites where they know my name and phone number.

Compiled programs do run faster than interpreted ones. In the lab we do a lot of batch processing, which takes time, and time saved means increased productivity.

Most accounting applications, today, have to be "interactive". Execution speed is important only to the extent that user

productivity is not adversely affected. (Read: If you can run faster than the operator, that is fast enough.) A recurring theme in this book is about how to achieve optimum performance from the interpreter.

Only the inept or uncaring write programs that run slower than need be, especially when alternative techniques cannot be fully rationalized by arguments about maintainability, or similarly subjective issues.

Brain strain is not an acceptable excuse to a professional. It takes intellectual effort to acquire knowledge. No argument. It seldom requires any extra labor to apply that knowledge, however. No confidence should be allowed for programs written by people too lazy to master their trade: Slow running code is often a reliable indicator of professional incompetence. This is equally true of all software, no matter what language it is written in. (It is true that programmer ineptitude is more conspicuous in interpreted BASIC programs. Sometimes.)

When speed freaks argue that they use compilers simply because interpreted programs are too slow, we can immediately assume that they are on somebody else's payroll. How long it takes to crank out an end product concerns them little. They get paid to come in every day and crank. Those of us whose income is in direct proportion to our productivity tend to see things from a rather different perspective.

When interviewing job applicants--I have hired, and fired, quite a few over the years--if they say execution speed is a primary factor in choosing a language, they do not get hired.

One more observation is offered about speed: An interpreted program running on an 8088 at 10 MHz may very well out perform a compiled one running on an old 4.77 machine. It is impressive even to me, to see two year old GW-BASIC programs running on a PS/2, racing along at 33 MHz.

After an emotional argument about performance, even if we give ground a little, we still must be adamant. QuickBASIC is not unlike a lot of other "modern" language products:

> It has not yet matured. The differences between releases are enough to make a wild man mad. In fact, we have gone back to using release 2. Later releases fixed some earlier bugs, but

also created more headaches in trying to write BASIC programs that will run in both interpreted and compiled modes.

All compiler writers are too dictatorial. Rather than just checking for syntax errors and translating our instructions, they try to force us to conform to their concepts of "good programming practices". They who have never had to write a payroll application, and support it year after year, are ill equipped to dictate programming doctrine. What we have to do, and how we do it, often differs from what is preached in the halls of academia.

The manuals are too thin. In addition to specifying what a program will do, software contracts have to specify what size box it will fit in, and what its performance thresholds are. Before the job begins. To have to do probe coding to find this out, in a compiled-language environment is too costly.

The QuickBASIC compiler is highly useful, even for programs that are not going to be delivered in compiled form. It is a super tool for finding coding errors that the interpreter may never bump into. The measly hundred or so for this tool can easily be offset by what a single on-site service call would cost, to correct a mistake that escaped your diligence.

The compiler manual is needed for reference, even for those writing only in GW-BASIC. Just as it is for us die-hards that cannot afford to cut the cord and go out into the world with nothing more in our pockets than QuickBASIC.

By reading both manuals, GW-BASIC and QuickBASIC, we can get two different author's definitions of things that were meant to be alike. The fingerprints of multiple authors can be seen in both books; there is some evidence however, they never read what each other had written.

The syntax of these two different BASIC languages is similar. So far. The key words that are common to both are spelled the same, and punctuation rules are alike. The grammar differs some: The compiler knows some key words that are totally incomprehensible to the interpreter, and vice versa. These variations are not too difficult to live with. Most of the differences in vocabulary cause no problems because, what is different is not useful in the "other environment", anyway.

Semantics is a harder nut to crack for bilingual programs. A

given line of code may run in either environment, but behave differently, because the two different language translators do not derive the same meaning from a given word or phrase.

Until release 4.5 of the compiler, internal processes produced similar answers. Presumably this is because compiled object code is still "interpreted" by a vast number of "modules" taken from the GW-BASIC interpreter. Run-time differences between these two pieces of software crop up because they interface to DOS differently, i.e., most of their differences relate to I/O operations and memory utilization.

The newer compilers are revolutionizing the language. Adding new gadgets did not hurt us old timers, much, until recently. Lately, we cannot even count on our programs to count the same. The adoption of IEEE numeric formats as the norm for BASIC is one more very good reason for staying with GW-BASIC, and for never becoming a wholehearted convert to QuickBASIC.

When they pull stunts like that, and "suggest" that we should back up and overhaul not only our old programs, but their data files as well, we are forced to revise our thinking about words like confidence and loyalty. Imagine a quarter of a million lines of code, running at a hundred or so installations, that maintain millions of records for all of those clients.

QuickBASIC proponents would like for us to believe that it is an enhanced or "extended" form of the language; those of us using the interpreter are laboring with a mere subset of the ultimate. If you swallow this without batting an eye, chuck this book and go read "Mein Kampf".

There is a risk of further divergence in these two languages. One indication that the "master plan" is aiming for a final decree of divorce can be seen in the respective manuals. The older, potty-trained versions of the GW-BASIC manual (1986) mentioned some of the differences in the two languages, albeit, somewhat sporadically. My newest manual, only a few months old, doesn't even bother to acknowledge its rival sibling.

The QuickBASIC manual continues to remark upon some of the more obvious differences in the two languages. Presumably to help us "up grade" from the interpreter. Which can be further read to mean they would like to coerce us in that direction.

Meanwhile, we can pray they have more social conscience than Adolph had. None of us are likely to begrudge them their profits, or that they are compelled to offer new products to keep their fiscal towers from toppling. At the same time, we hope they will not ignore what happened to the Avanti and the Edsel. The masses may be ignorant, but some of us peasants are not as gullible as those on high might think.

So much for the Whom, Which, When and, Why: From here on it is all about how. How the interpreter works, mostly. And that is based on how I perceive it, looking from the outside in, as a user of GW-BASIC. My sporadic notes about some differences between the interpreter and QuickBASIC have no ulterior motive; those that are mentioned are those that caused me grief because they are not documented anywhere else, as far as I know.

When QuickBASIC is mentioned hereafter, it refers to release 2. Nobody could document all of the differences that have occurred since. It often takes me a year or two find most of the bugs in each release. New releases are being issued so rapidly today, there is not enough time to even read the manuals for any one, before it is time to start all over.

What follows is not a tutorial on how to program. In BASIC or any other language. Teaching is properly the province of those who know how to teach.

What follows is simply a memory dump. Of my memory. Of my experience, and how I make use of what I have learned. And some of my code. If you already have a better wheel, good. If not, some of what works for me could save you having to reinvent solutions to programming problems that have existed for years.

Expert? Hardly. Read me as a coworker, passing along to my fellows what I know (or think I know) before I am too old to remember it all. When I am guilty of a mistake, or awkward phrasing, remember that I am simply a simple programmer. (I nearly drowned the last time I tried to walk on water.)

Chapter 2 = PROGRAMS

A program is stored, physically, as a file. Logically it is organized as what is traditionally called a sequential file. That is, a file of records (lines) of varying lengths, one record following another, sequenced in 1-2-3 order.

If you save a program as an ASCII file (SAVE "program",A) it is in fact output according to the conventions that have by now become known as "...a standard sequential file."

Each line of your program is a variable-length record. The last two bytes of each record are a CR/LF pair (a carriage return and a line feed). The end of the file is marked by a single-byte (control-Z code) immediately following the last record. All bytes, in all records, are standard ASCII character codes. (Save those above CHR$(127), which are not truly ASCII.)

As Mr. Holmes would say: "Elementary, my dear Watson". This much is learned easily from the manuals. And at some point, most novice programmers write experimental programs to "read" program files. Some progress rapidly to the next step and write programs that "write" programs.

Program generators, as a concept, is as old as is the business of programming itself. The first time any programmer with very much experience has to write a couple of hundred lines that look-a-lot-alike, he is very likely to write a "tool" to generate those lines automatically.

By the time most students have gotten this far they also begin to wish they had more tools. And veterans of other programming languages are very quick to notice the conspicuous absence of tools in this environment. At least part of that void can be filled by Chapter 15 which contains handy routines from my own toolbox.

This chapter seeks to fill another void. When your world dictates the need for custom-made tools, that have to work on programs as they are in memory, or as they are in files that were saved as "binary", what is in those bytes must be known. That nitty-gritty detail is provided here, but not just for the benefit of tool writers.

Suffer my favorite contention repeated often elsewhere: To be

able to write programs that run as efficiently as possible requires an understanding of how the interpreter works. This narrative can be read to further that insight, savoring the general concepts, skipping quickly over the gristle.

A brief preamble is necessary before wading in. The more explicit a technical note is, the more apt it is to be wrong. Not because of errors (which are certainly possible), but more likely because we are not viewing exactly the same thing. The world is constantly changing around us, and that cliche is so very applicable to the world of programming. On the off chance you encounter a bent byte in what follows, perceive it as a mere pebble in a swift stream: Wade on.

A program is stored, physically, as a file. Logically it is organized as what is traditionally called a sequential file. That is, a file of records (lines) of varying lengths, one record following another, sequenced in 1-2-3 order. This repetition is still basically true, but from here on, when you save a program without the A-for-ASCII option, it is a whole new ball game.

A program file is, essentially, what is sometimes called a mirror-image memory dump to disk. That is why LOAD and SAVE type activity is fast: Reading and writing are done on the basis of physical blocks, not at the logical line, or record level. While in memory (and therefore while on disk) a program is still internally organized as discrete lines, arranged in line-number order, just as you see them with LIST or LLIST.

Granted, this much can be learned by a careful reading of most manuals. Some mention that the lines are stored in a compact, compressed, or "tokenized" form. Which is about as far as any of them go. Unquestionably, the interpreter program is a very sophisticated, highly complicated, special piece of software. But, it is still a program. It "processes" your program. It starts where you do, on the first line, examining each of your statements, doing what you tell it to do, one step at a time.

This grossly understated, oversimplified definition of the interpreter is the fundamental perspective from which to read what follows. The purpose at hand is to view what is in the program lines themselves. Seeing that, we can often surmise what the interpreter has to do. How it actually does it can remain obscure.

All lines begin with the first four bytes having the same order and purpose, and end with a byte equal to CHR$(0). The first two bytes are an address pair. They contain the actual address of the start of the next line. Starting with the next two bytes, the pair that contains your line number, everything within a line is in exactly the same order as it is when it is viewed as an ASCII-text line. Therein ends its similarity. The bits in the bytes themselves are formulated to suit the interpreter.

Some bytes still coincide with the ASCII character set, and their interpretation remains unchanged. Because any given byte may range the full 8-bit spectrum of 0-255, in decimal numbers, most of them are bound to look like printable characters. But a byte that looks like a CHR$(65) may not be for the letter "A", at all. It may be a code, or part of a code, or a number, or part of a number, or....

The bytes within a line are parsed as 1-byte codes, or as words (groups of bytes). A word may be as small as 1 byte. Some are 2, some 4, and naturally, some are 8. Grab on now to that thread that is woven throughout machine language programming: 8-bit bytes used as 1, 2, 4, or 8-byte words.

Nearly all of the so-called "key words" in BASIC are stored as tokens (codes). When you type a line of a program using the BASIC editor, the text of what you type is translated. The key word BEEP is stored as a single byte, for example; it looks like a CHR$(197). When you see BEEP, a token that is equal to the number 197, in decimal, has been translated back into the four upper case ASCII letters that spell BEEP.

> Time out. From here on assume all my numbers are decimal. To keep using phrases to ensure that 197 is understood to mean a byte equivalent to CHR$(197) is redundant. It slows down your reading, and my writing. Now to resume....

Many key word tokens are, effectively, two-byte codes. The first byte serves as an indicator that the token in the next byte is from an alternate translation table. The token for SWAP is 164 and the token for LOC is also 164. So, a token byte of 164 is either for SWAP or LOC, depending on which table is used for translation, i.e., on whether or not the token is preceded by a 255-table indicator byte.

In all, there are four key word tables. The single byte 129-token translates to END. A 255-byte followed by 129 is translated as LEFT$, a 254 then a 129 is for FILES, and 253 followed by 129 means CVI.

Because the first byte dictates how the token that follows it should be translated, we can get an immediate insight into how the interpreter actually works. It processes a line from left to right, one byte at a time. On the basis of what a byte has in it, it can proceed at the rate of one at a time or, gobble up 2, 4, or 8 bytes for its next trick.

When a program is "running", when the interpreter bumps into a 253-byte, for example, it knows that the next byte is a token, and it will be a function call to do CVI, CVS, CVD, MKI$, MKS$, MKD$, or EXTERR (because only 7 unique tokens are expected to follow a byte that contains 253).

Notice how "a byte" indicates what should be expected next. To be able to parse a line--to separate it into lexical units words)--merely requires an algorithm that mimics the logic of how the interpreter does it. And that is not very complicated at all.

Constructing a tool that will translate the number 145 into the word PRINT, for example, requires no great effort. What it takes in the way of routines to carry out a command such as PRINT can take many hundreds of program lines. Without seeing those lines, or even knowing machine languages, we can appreciate the long hours and hard work that went into the writing of the interpreter itself. By stepping along a line, a byte at a time, just as that program does it, we can grasp the basic principles by which it works, however. Like this:

Begin at the beginning. The first line of your program. The first two bytes are an address-pair. To get their decimal value, if need be, multiply the second byte by 256 and add to that, the value of the first byte. And take a note. This same arithmetic feat can be used to convert all 2-byte words that represent addresses or line numbers. (Which are stored in the way machine language works, i.e., backwards compared to how we would do it in our head.)

Add 2 to your byte pointer. The next pair of bytes is also a 2-byte word. Do the arithmetic. 'Lo and behold, the result

is equal to the line number you used when this program line was created. Now increment your byte pointer by one and get ready for some real fun.

If the next byte is 32, 44, or 58, it is a space, a comma, or a colon, same as in ASCII. Bump on. If the next byte is zero, you have reached the end of the line. If it is greater than 128, it is a key word token. If none of these tests are true, you are now looking at something that you made up--a literal, a constant, or a variable name--or, your pointer is in the wrong place. Or you are trying to read somebody else's mail.

By this time, if you are actually writing a programming tool, what you want to see most are the tables at the end of this chapter. But don't start coding yet. It would be useful to know how the information in these tables was compiled. (In case your version of the interpreter differs from mine.) And, there are some tidbits that need to be known that are not obvious when looking at charts alone.

Speaking of charts: In days of yore the manuals always had a chart that listed all of the "reserved words". Which was handy to review, to keep from inadvertently creating a conflict when coining variable-names. My newest (seldom-used) manual has no such chart. It does say, "All GW-BASIC commands, statements, functions, and variables are individually described in the GW-BASIC User's Reference." Poppycock. Attempt LCOPY = 1 and you will get a syntax error. LCOPY 0, on the other hand, gives no error, but neither does it do anything.

> LCOPY is a truant command. It worked in only one release--I forget which one, 2-dot-something--but it now simply works like a no-op. Perhaps that is why the books no longer list all of the reserved words; they would have to say, exactly, on what day of the week it could safely be consulted.

To find out, exactly, what token is used to represent a given key word, type it as the first word in a BASIC program. SAVE it, then use DEBUG--or some other tool--to see just what the interpreter converted that word to. By the way, the very first byte of a "saved" program file is a file-type indicator. (It may be 255, which indicates a "normal" BASIC program; 254 says it was saved with the "protect" option.) Remember also, to jump over the first four bytes at the beginning of each line before you start looking for a key word token.

PS: A first byte of 254 or 255 does not always mean that what follows is a BASIC program; it is simply what the interpreter looks for to "know" if the file you are loading is a BASIC "tokenized" program.

If you don't know all of the key words built into your version of the interpreter, there is a way to find out. But it is not particularly easy. Unfortunately, they are not all shown in all manuals, and some, although in a manual may not be in your software. Back to DEBUG. Dump the interpreter itself. Look for what resembles BASIC "reserved words". They are not stored as pure ASCII; the first letter of a word and the last letter, or both, may look like mumbo jumbo, but the letters in the middle of the longer ones like RANDOMIZE are still recognizable as ASCII upper case letters.

More than just accurate tables of key words and their tokens are needed. Here are some other interesting things to expect when parsing BASIC lines. And some more insight into how the interpreter works when it is executing a program.

Numeric literals are stored in a line in exactly the same format as they would be as if you had assigned them to the least precise variable that would be required to hold them. For example: -32000 is stored in two bytes (exactly the same as in an integer variable). This in-line literal is preceded by a code-28 that indicates that what follows is a 2-byte word, and that it should be translated as an integer. See now why it happens that, although you typed A=99999, later you will see A=99999! when you do a LIST. (Integers do not get a free appendage but all larger numbers do, or they play back as if you had typed them using pseudo-scientific notation.)

This same concept is true for all in-line values. They are stored in ready-to-use format. No conversion is necessary. The interpreter can grab a 1, 2, 4, or 8-byte word and use it instantly, just as it is. It's better to do the conversion while you are typing. You won't even know when it is done. (Not many typists can outrun a modern micro.) See also why a MERGE can take awhile: There's a whole lot of converting going on during the load process.

There is one type of conversion that does take place during execution of a program. If you are poking around in program memory while a program is in progress, watch for this one.

GOTO, for example, is followed by a line number. Line numbers are stored in 2-byte words. They are normally preceded by a code-14 byte. When the interpreter bumps into the 14, it runs through your program to find the line that has the matching line number. Now the tricky part: The three bytes after the GOTO get changed. The code will be changed from 14 to 13, and the real address of the target line will overwrite the two line-number bytes. Once found not forgotten is the moral.

Converting translated addresses back into line numbers can be done very quickly, by the way. If you bump into a code-13, use the address following it to get the unchanged pair of bytes from the beginning (+2) of the target line, and change the lead byte from 13 back to 14.

Most of the time your variables look just like you typed them. As ASCII upper case letters, numbers, and appendages. One exception is the case of a user-defined function name. The FN itself is converted to a 1-byte token (209), but the rest of your name remains unchanged.

Confusion--and not a little aggravation--can arise when you are parsing for variables only. Not all key words are tokenized. There are only a few that are not, but because they are stored internally as ASCII letters, they have to be parsed as if they are variables. Then you can decide whether you invented the name or BASIC did. The ones that are, and those that are not key words tend to be different sometimes. (If you call your machine Junior, watch out for PALETTE. The seniors use a token, but some PC Jr's do not.)

Another perversity: B and BF can certainly be variable names. But they may also be "un-tokenized key words". If they follow the second comma in a graphics LINE statement, they are merely switches to condition how that statement is executed. To find all variables, only, accurately, your algorithm will have to become context-sensitive when it encounters the 176-token for LINE. A 133-token soon after means you found LINE INPUT and any subsequent B or BF are variables. Otherwise, the letters B or BF--following a comma-count of two--can be skipped.

DATA statements are always whole lines. If the first token following a line number is 132, the next byte should be 32 (a space character) and the rest of the line is pure ASCII up to the final byte (which is always zero). Notice too, numeric

DATA elements are not converted to internal format until the moment that you do a READ. Another performance hint, albeit a rather small one. (Reading strings of literals from DATA statements is not an efficient way to program.)

Remark statements are interesting, and a little odd. A token of 143 translates as REM, but only if the next byte is not 217. The two in a row--143 and 217--translate as an apostrophe, the shorthand symbol for a remark. (And this pair is followed by an arbitrarily imbedded code-58--a colon--for some obscure reason.) Beyond that, whatever you typed is stored just as is, as ASCII characters. Notice also that the shortest remark is at least two bytes. If you use REM it is stored as 143 followed by the syntactically required space character (32). Two bytes. If you use the apostrophe, it is stored as 143, then 217, then 58, but no space character is required. Three bytes. So, REM is less costly than the apostrophe (but is less pleasing visually).

Another performance note: Both DATA statements and remarks are, effectively, do-nothing bytes when they are encountered by the interpreter while it is executing a program. To get from the token to the start of the next line, the interpreter has to bump along, one byte at a time while looking for the zero at the end. (It forgot, apparently, what address is in the pair of bytes at the beginning of this line.) So, use remarks freely, but put them only after statements that have an emphatic conclusion. After NEXT instead of after FOR, for example. And never intersperse DATA statements in a stream of executable lines unless you like programs that run slower than they need to.

Although the token for ELSE (161) comes from the single-byte tokens table, it is always preceded by a 58, which is normally seen as a statement separator. So, when you bump into a 58, look at the next byte before assuming that what is coming up is the next statement on a multistatement line. (Normally ELSE should only come after THEN, as we all well know.)

Quotation marks are also a little odd. They are stored as 34, same as in ASCII, but they are supposed to be used in pairs. If, for example, PRINT "hello" is encountered, the first quote turns-off the tokenizing. The next one turns it back on. So, everything that you bracket with quotes gets stored just as you typed it. And if you failed to type a second one, everything from the first quote through the end of the line is treated as one continuous string of text. Which explains why you see some funny stuff, sometimes. (A missing quote can be the cause of

some not so funny bugs.)

Parentheses, on the other hand, must be used in matching pairs, and they are stored as codes 40 and 41, respectively. Or you will trigger an error trap. The LEFT$ token, for example, will (should) always be followed immediately by a code-40 byte.

Another aside: See from the above why an error trap can sometimes be confusing. You confused it. Parentheses and other "syntax characters" are fundamental to the business of parsing a line. Some codes indicate that the byte pointer should jump forward a specific number of bytes. If the code found at that point is not what is normally expected, it can be assumed that whoever typed that line was not playing by the rules. The best "error" that can be given is based on what the pointer is now seeing. Bytes bumped over are history.

And some bytes ought to be history. Bytes that really are bumps are not unlike the speed bumps in parking lots. They slow down your program. Not always much, maybe, but if you like programs that run in the fast lane, omit anything that is "optional". Many times the third argument in MID$ expressions can be omitted. The pound-sign can almost always be omitted. It is a must before the file number in INPUT, and PRINT, and WRITE statements. But the rules are inconsistent when a file number is used between parentheses. Like in VARPTR(#1) the rule is different than for LOF(1). Which is not the only inconsistency about parentheses.

For some odd reason the left parenthesis is not stored in a line in two cases. The key words TAB and SPC have their trailing appendage imbedded in the translation tables. Their tokens will not be followed by a code of 40. And these two are perverse in another way: They can be used only in some form of a PRINT statement. Presumably these genetic traits have something to do with their heritage.

A number of family characteristics are noticeable in the key word translation tables. Most of those in the first family are commands (as opposed to functions). Most of these kids are not expected to have parenthetical expressions tagging along behind them. The tokens in this family range from 129 to 244, with a few gaps. Some of the gaps are caused by infant mortality--key words that used to be in BASIC but are no longer with us. And some are recent adoptions, words like COLOR, that have been added as the language has grown up over

the years.

The second family is a little more purebred. This is the gang guarded by a 255-byte. All of these key words are always followed by a left-parenthesis, except the word PEN. (A misfit cousin, no doubt. Depending on how it is used, PEN may or may not have a code 40-byte tagging along behind it.) The 37 kids in this clan are numbered from 129 to 165, and none are missing. Which implies, in the absence of family planning, up to ninety more (166-255) could come along in some future generation of the language.

The next family down the line, guarded by a 254-byte, looks like an orphanage. This group of 27 tokens range from 129 to 155, with no gaps. Most of this bunch are relative newcomers; especially those that provide an interface to the operating system. Still, the older ones near the top of the list have been around since the juvenile versions of "disk BASIC".

For a long time there were only six members in the fourth family, the one guarded by a 253-byte. These three sets of twins were originally conceived to be useful for working with so-called fielded-variables. (They are not restricted to that playground, however.) Then along came EXTERR. Wrong bus, maybe? Notice the empty seats.

Had this chapter begun with WHILE, we are now nearly ready for WEND, and a fall-through to those tail-end tables. Remember those two bytes at the beginning of each line that address the start of the next logical line? That address is accurate only while a program is memory resident. When you SAVE a program, those addresses get saved, right along with everything else. When you LOAD, however, just where the file is placed into memory at that time may be different than it was the last time it was used. During a LOAD (or a RUN, or a CHAIN) the interpreter must recalculate all of those addresses.

Notice that the address values are proportionally correct in program files stored on disk. A single addition or subtraction factor can be applied to them all, to maintain their chain-to relationship. Equally, the difference in the address headers of two successive lines can be used as a byte-count of the length of a line.

If you are reading a program as it sits in a file, the line

addresses shown are those that were, once upon a time. If you are peeking at a program in memory, you are seeing things as they are now. Either way, now we know how to see a program just as the interpreter sees it. Even if that vision is still a little fuzzy, this overlook will, hopefully, broaden the horizon.

| Internal Code Assignments |

0	End of a program line
1-10	Not used (should not be encountered)
11	Translate next 2 bytes as Octal, like &O1024
12	Translate next 2 bytes as Hexadecimal, like &H7D0B
13	Next 2 bytes are the address of another line
14	Translate next 2 bytes as a line number
15	Translate next byte as a numeric literal (0-255)
16	Not used (should not be encountered)
17-26	Translate this byte as a decimal digit (0-9)
27	Marks end of file (preceded by a zero-byte)
28	Translate next 2 bytes as an integer
29	Translate next 4 bytes as a single precision number
30	Not used (should not be encountered)
31	Translate next 8 bytes as a double precision number
32-127	Translate as standard ASCII text characters unless 58 is followed by 161; translate this pair as ELSE
128	Not used (should not be encountered)
129-252	Translate as a key word from table 1
253	Translate next byte as a key word from table 4
254	Translate next byte as a key word from table 3 If 1st byte in file, it was saved with P-option
255	Translate next byte as a key word from table 2 If 1st byte in file, this is a LOAD-and-go program

| Table 1 = Single-byte tokens |

129 END	151 DEF	176 LINE	201 KEY
130 FOR	152 POKE	177 WHILE	202 LOCATE
131 NEXT	153 CONT	178 WEND	204 TO
132 DATA	156 OUT	179 CALL	205 THEN
133 INPUT	157 LPRINT	183 WRITE	206 TAB(
134 DIM	158 LLIST	184 OPTION	207 STEP
135 READ	160 WIDTH	185 RANDOMIZE	208 USR
136 LET	161 ELSE	186 OPEN	209 FN
137 GOTO	162 TRON	187 CLOSE	210 SPC(
138 RUN	163 TROFF	188 LOAD	211 NOT
139 IF	164 SWAP	189 MERGE	212 ERL
140 RESTORE	165 ERASE	190 SAVE	213 ERR
141 GOSUB	166 EDIT	191 COLOR	214 STRING$
142 RETURN	167 ERROR	192 CLS	215 USING
143 REM	168 RESUME	193 MOTOR	216 INSTR
144 STOP	169 DELETE	194 BSAVE	217 ' (rem)
145 PRINT	170 AUTO	195 BLOAD	218 VARPTR
146 CLEAR	171 RENUM	196 SOUND	219 CSRLIN
147 LIST	172 DEFSTR	197 BEEP	220 POINT
148 NEW	173 DEFINT	198 PSET	221 OFF
149 ON	174 DEFSNG	199 PRESET	222 INKEY$
150 WAIT	175 DEFDBL	200 SCREEN	---
(gaps: 154, 155, 159, 180-182, 203, 223-229, 245-255)			
230 >	234 -	238 AND	242 IMP
231 =	235 *	239 OR	243 MOD
232 <	236 /	240 XOR	244 \
233 +	237 ^	241 EQV	

| Table 2 = Key word tokens preceded by 255 |

129 LEFT$	139 EXP	149 ASC	159 FIX
130 RIGHT$	140 COS	150 CHR$	160 PEN
131 MID$	141 TAN	151 PEEK	161 STICK
132 SGN	142 ATN	152 SPACE$	162 STRIG
133 INT	143 FRE	153 OCT$	163 EOF
134 ABS	144 INP	154 HEX$	164 LOC
135 SQR	145 POS	155 LPOS	165 LOF
136 RND	146 LEN	156 CINT	
137 SIN	147 STR$	157 CSNG	
138 LOG	148 VAL	158 CDBL	

| Table 3 = Key word tokens preceded by 254 |

129 FILES	139 COMMON	149 ERDEV	159 PALETTE
130 FIELD	140 CHAIN	150 IOCTL	160 LCOPY
131 SYSTEM	141 DATE$	151 CHDIR	161 CALLS
132 NAME	142 TIME$	152 MKDIR	162 ---
133 LSET	143 PAINT	153 RMDIR	163 ---
134 RSET	144 COM	154 SHELL	164 ---
135 KILL	145 CIRCLE	155 ENVIRON	165 PCOPY
136 PUT	146 DRAW	156 VIEW	166 ---
137 GET	147 PLAY	157 WINDOW	167 LOCK
138 RESET	148 TIMER	158 PMAP	168 UNLOCK

| Table 4 = Key word tokens preceded by 253 |

129 CVI	132 MKI$	135 ---	138 ---
130 CVS	133 MKS$	136 ---	139 EXTERR
131 CVD	134 MKD$	137 ---	

| Key words not tokenized (kept as ASCII) |

ACCESS - - - as in OPEN ... FOR RANDOM ACCESS
AS - - - - - as in OPEN ... AS --- and --- FIELD ... AS
ALL - - - - as in CHAIN ... ,ALL
APPEND - - - as in OPEN ... FOR APPEND
BASE - - - - as in OPTION BASE 0 --- or --- OPTION BASE 1
OUTPUT - - - as in OPEN ... FOR OUTPUT
RANDOM - - - as in OPEN ... FOR RANDOM
SHARED - - - as in OPEN ... SHARED
SEG - - - - as in DEF SEG

Chapter 3 = VARIABLES

Some texts would aptly lead off here talking about "heaps". (When I was in school, my car was a heap.) My preference is more accurate: Variables, their contents and the names by which you know them are organized internally as tables. The word table connotes logical structure....

Two tables may exist. One for simple variables, another for arrays. These tables are maintained in memory immediately following your program. If your program has both simple variables and arrays, the first table holds simple variables, the second table is for the arrays.

Both tables are maintained and searched independently. Their structures are similar, however, and additions to either table are done on much the same basis.

As your program executes, any statement that assigns a value to a variable causes one of the tables to be searched. It is always done sequentially, from the top of the table downward. In the event a search fails--in the event this is the first time a variable has been assigned a value--its name is added to the bottom of the table.

Because the simple variables table is first, each addition to this table means that the entire second table must be shifted downward to make room for the name being added to table #1. Ergo: This is why most manuals mention that you should do a VARPTR immediately before any attempt to directly address array-bytes in memory--in case the arrays have been moved because of a recently added simple variable.

What is seldom made clear, however, is just when it is a name is added to either table. Some manuals are so erroneous as to say "on any first reference" to a variable name. And I have read none that fully explain about deleting names from the tables, nor how searching for variables is actually done.

This narrative is needed to fill the literature void. Run time performance is why. Most of us have heard that BASIC programs run too slow. Often times they do because many programmers have little concern for writing efficient code, or they are simply unaware of how the interpreter works.

The last part of this chapter covers what is actually in the

variables tables. Most manuals give only a modicum of this detail. What is here can improve your productivity when, if you need to know specifics, you need not have to find out the hard way.

Applying what follows can make a dramatic difference in how fast some programs run. In any event, coining names, and when in a program to first declare them, ought to be decided deliberately: Based on a full awareness of cause and effect.

When names are added: It is done only by memory allocating statements. Meaning on a LET, either explicitly stated or implied as in A = 3. Or, in the case of arrays, on a DIM, explicitly stated, or not. They are added only once, and remain in place in the table--relative distance from the top--during the continuous execution of a given program. (Remember, arrays are not static. More on that later.)

Oddly enough, an LSET or RSET works as an explicit declaration, even when the consequence is a null variable. Like, if you do LSET X$ = "hello", and this is the first time X$ has been named, the name is added to the table. Now do a PRINT X$. It will read as if you had done X$ = "". From this we can deduce no distinction is made for fielded variables vs. "regular" string variables during the mechanics of searching for names, and when adding them to the tables. (There is a distinction about where the data itself is stored. See Chapter 4 for that greasy grit.)

The theme of explicit vs. implicit memory allocation must be carried one step further. When a user-defined function statement is executed, such as DEF FNA(G) = B*C+G, the name of the function (FNA) and its parameter (G) must already be in a table, or they are added at that time. Although no data is actually acted upon, yet, space must be allocated now for the address that will point to this function-expression, and for its parameters. But notice that its arguments, B and C, are not looked for at this point. The search for them is not done until (unless) this function is actually made use of in some statement or expression later on in the program. This is only one instance of a variable "being referenced", but it is not added to the variables tables.

When names are not added: Some "references" to variable names

require no table searching, hence they have no influence on the ordering of table entries. Default typing is such an instance. The statement DEFINT C-L assigns no data to storage, thus, no table searching is done. It matters not a whit, where in the program this statement is encountered.

Dummy variables are another case in point. In the function POS(B), for example, the "B" is not logically needed, i.e., what is or is not in that variable, or whether or not it even exists is unimportant. So PRINT POS(B) causes no search, and so, has no influence on names-tables.

The naming of variables in a COMMON statement is another case. Merely naming them causes no table search, so it has no side effects at that point. Table searching and the addition of COMMON names to the tables works the same as for "uncommon" names: When they are used in a way that demands reference to some area in working storage. Chapter 11 covers the use of COMMON and CHAIN in depth. Here, note that in a chained-to program, COMMON names are passed along as is, ordered the same as they were stacked by the program(s) that ran previously. New additions to the tables caused by the program now running are added to the bottom of the tables that were inherited.

Searches can be provoked sometimes without causing any names to be added to the tables. And this is almost a worst-case coding instance; the performance overhead for doing the search must be suffered with no global benefits to be derived. For example: IF M$ = Z$ THEN.... In the event Z$ is null, i.e., it has never been assigned any data, the entire names-table must be searched to learn that the name does not exist (and therefore, Z$ = ""). But this will not cause Z$ to be added to the table. This same consequence is true when doing WHILE/WEND and ON GOTO types of conditional tests, by the way, and is also true for tests involving numeric variables that are zero simply because they have never been assigned a value since RUN.

To this point then, take note that the tables are built in sequential order: The first-named variable will be first in the table, the second-named one will be next, and so on. Table searches are always done from the top, looking for the one just named. If it is found the search is ended. If it is not found --the end of the table is reached--that name is added, but only if the usage expression dictates that space be allocated to hold literals, constants, or address pointers to be used (maybe) later.

Infer now, also, why it is possible to use the same name for both a simple variable and an array. Two tables. Using just "A" causes a different table to be accessed than an A followed by a left-parenthesis.

Not so easily inferred is the fact that every instance of a variable name, in an expression, instigates a full top-down search. This, despite the high frequency of repetitious naming in BASIC. Like, A = A*A+3. Every encounter of "A" causes a search, even though the last previous search was for the same name. So, A = X+A+7 is just as efficient as--or is just as inefficient as--A = A+7+X. Both cases require three name searches. By the way, see why in-line literals are faster than variables. As the interpreter steps through an expression like the last one, it does not have to search for the "7". It just ran into it.

On the other hand, no repeat-search is needed for the control variable in a FOR/NEXT loop, unless that variable is renamed after the NEXT. When FOR I = 1 TO 10 is executed, the name "I" must be searched for only once. Pointers are then pushed onto the interpreter's stack to control the loop. Some champions of pretty code insist on renaming the control variable after every NEXT. Like NEXT I or NEXT B, or what have you. Agreed, it is nice to read. But, it slows down your program.

Every time NEXT is encountered a table search must be done to ascertain the pre-existence of that name in the table. And to confirm that it is the same name used in the last FOR, by the way. Read that phrase, "the control variable" closely: It means the variable used with FOR or NEXT. Expressions that use that same variable inside a loop are treated no different than variables used anywhere else in a program.

> Ergo: Omitting the name after NEXT optimizes performance. Using a single NEXT as the terminal point for nested loops requires the naming of all FOR-variables, in proper order, after NEXT. (Which is slower than NEXT:NEXT, for example.)

The parsing of variable names: As the interpreter bumps along a line in your program an ASCII upper case letter (A-Z) denotes the start of a name. The name-string continues until a byte is encountered that is not a letter or a digit or a period. If the next (one) character is one of the data-typing appendages (%#$!) it will be used in the table search, but the appendage

itself is not stored, as such, in the tables. And finally, if the next character is "(", it too will be used (but not stored) to determine which table to search.

Table selection: Logically, the two tables are distinct. If the name in your program-text has a left-parenthesis as its last character, the array table is searched. Any character denoting the end of a name other than "(" causes a search of the simple variables table.

How table searching is done: Ostensibly, by comparing names. In fact, it is a little quicker than that. The first test is for a matching data-type name. All variables that are not of the same type as the one being searched for are stepped over immediately. The next test is done on a name's first two characters. If they are not matched, searching continues. When both the data type and the first two characters are alike, a length comparison is made. Searching resumes immediately when unequal lengths are found. Otherwise, a comparison is done on a byte-by-byte basis until a difference is found, or until the end of the name is reached.

Searches for names of only one or two characters are quickest. Next quickest are longer names that have the same first two characters, but are of unequal length. The unwary that adopts naming conventions such as

```
TOTAL.A1   ...   TOTAL.A2    ...   TOTAL.A3
```

may be surprised at the better performance obtained with

```
A1.TOTAL   ...   A2.TOTAL    ...   A3.TOTAL
```

with no real loss of mnemonic quality. Granted, the first example is (maybe) more aesthetic. Which is needed more, pretty code or efficiency should be thought about, rather than build-in less than optimum performance on a whim.

How memory space for the two tables is managed: Because the fundamental structure of the two tables is similar, the above comments about when and how tables are searched, and when and how names are added apply to both simple variables and arrays. Table-space management is considerably different, however.

The first table, the one with the names of simple variables

in it, grows longer as a program runs, but it never gets shorter. (Exception: CLEAR erases everything, of course; this is one of the least useful, and certainly one of the most ill-defined key words in the language.)

The second table grows longer, also, but an ERASE will cause all of the bytes in all of the arrays that follow it to be moved up, to overlay the area just "erased".

Consider: A handy and sometimes quick method of clearing totals is to ERASE and again DIM the same array. Notice the potential for reordering the relative position of array names. If a DIM is done in the initialization phase of a program, for several arrays, their position in the table is consistent with the order in which they are first named. An ERASE done on one, followed by another DIM of the same array pushes it to the bottom of the array storage area: It is now at the bottom of the list for future searches.

Significance (an example): To make a sort loop run faster it is better if the array's name is at the top of the table. Because repetitive references to this name must be done, each name-search in each iteration of the loop will take less time than would be required if the name were at the bottom of the array-variables table. If you clear an array--using an ERASE and DIM sequence--to ready it for sorting another batch of data, know that the second sort may take longer than the first one did.

It is reasonable to assume that the interpreter can do a block move of the array storage area on an ERASE much faster than a FOR/NEXT loop can clear array elements. In many applications, total system throughput may actually be better in the second case. The seemingly longer time needed to do discrete variable assignments to clear array elements can sometimes be more than offset by maintaining the array's name at the top of the table. The factors to consider are: How many unique array names are in the table, how many times a given array has to be cleared for reuse, and how much searching for array names is involved in each iteration of a looping process.

> Note: ERASE and DIM done on string arrays portends a serious performance risk beyond the simple issues described above. (Chapter 4 is a full discourse on how the interpreter manages string space.)

To what extent the points made thus far should influence the design of a given program is a function of the end purpose of that program, and what its overall design requirements are. Strong arguments can be made for "pretty code", for programs that must be modified periodically, and especially for those that may have to be worked on by more than one programmer.

All such arguments are invariably subjective, however. What follows is an attempt to reduce performance considerations to an objective level. Recognize that what follows must still be viewed as approximate, or "rules of thumb", because of the infinite variety that can occur in the world of programming.

Another preamble is needed before getting to the meat of the matter. Speed is relative. If your machine runs at 8 MHz and mine at 16, then mine can be assumed to be twice as fast as yours. For the purpose needed here. Different kinds of micros, and memory chips, and machine configurations, and so on, all do impact the truth of claims about one being such and such faster or slower, etc. Still, on a given machine, if one coding technique takes 20 seconds and another technique takes 10, we can jump to the conclusion that the faster technique will still be about twice as fast as the alternative on some other machine. (And assuming, of course, use of the same versions of the interpreter and other "systems software".)

Given these assumptions, the following discussion makes use of Time Units (TU) for comparison purposes to preclude my being misread as having said such and such takes so many fractions of a second. My research did involve taking clock times, but neither of us should have to do mental gymnastics because one machine is supposedly 53.7% faster than another. Ad nauseam.

If we accept a base line for the time required to search for a 2-character variable name, if it is at the top of the table, as 1 Time Unit, then for one located at position 100 in the table, the search takes 2.2 Time Units. The additional 1.2 TU seems so small, especially if we digress for a moment and think of it as 1.2 milliseconds. In practice, all clock ticks are important. For highly repetitive processes they can be very significant, no matter how fast the clock is ticking.

Here is a line of code from a sort algorithm:

```
IF AX(IC)>AX(IC+1) THEN SWAP AX(IC),AX(IC+1):EX = 1
```

Count the number of table-searches required if the SWAP must be made. Four for AX(, 4 for IC, and 1 for EX. So, 9 table searches are required, for this one line of code, every time it is fully executed, in each iteration of the loop.

In a bubble sort of 1000 elements, 100% out of order, one pass through the table would take 9000 TU just for searching for the variables. And this is true only if IC and EX are the first two simple variables, and AX(is the first array name. Thus, it is no wonder sort routines run so slowly.

If, on the other hand, IC happens to be way down in the list, say at position 100, this same sort would require an additional 4800 TU for each pass through the loop. And if EX is also way down in the search list, say at position 99, another 1200 TU would be required: An additional overhead burden of 6000 TU.

Fifteen thousand vs. nine thousand. Holding to my theme that speed is relative, this looks like you could chop six minutes off of a fifteen-minute sort. Nice. In fact, this one would run nearly twice as fast because the first pass runs 999 times, the second runs 998, the third runs 997, and so on. (In a bubble sort, the depth of the loop is shortened on each full pass by at least one. The name-search overhead is a constant, for each line executed in the loop. As the loop gets shorter, the burden is greater proportionally in terms of the amount of work being accomplished.)

We would all like to cut sort times in half. But, don't start recoding old programs yet. The example used here is heavily loaded for dramatic affect. How many times does a sort have to be done on data that is 100% out of order to begin with? How likely is it your loop's data variables are so far down in the names-table? One hundred simple variables is a lot in any of my programs.

One opportunity can be gleaned at this point: If some of the variables used in a long-running loop are named for the first time late in the program, performance would have to be better if they were named earlier. Even if that declaration seems redundant, out of place, or has no obviously logical purpose.

My average program has maybe 40 variable names, rather than the fictitious 100 above. So, the potential for improving run time performance by cleverly stacking the names-tables is much less. Still, the possibility of shaving even 10 or 15 minutes off of a one-hour run is well worth some serious thinking about where

in a program to initially declare frequently used variables.

What about long variable names? They are easier to read, no doubt. If we deliberately contrive a program having 100 unique variable names, all of equal length, like:

```
TOTAL.A0   TOTAL.B1   TOTAL.C2 ... TOTAL.V3
```

Then 1.16 TU are required to find the first name. And 6.3 TU are required to find name number 100 in the list. There really is a difference, as we knew, intuitively. In this (ridiculous) case the last named variable takes nearly three times longer than would be the case if all the names had only 2 characters.

Agreed, this contrivance is farfetched. But it does support my contention that A0.TOTAL, B1.TOTAL.... is more optimum than those above. And 1.16 TU vs. 1 TU is the proportional penalty for a name with eight characters vs. a name with only two. Read this carefully, however. The penalty applies to the time required to search for a given variable. Which may or may not have any real significance on program performance overall.

How to get optimum performance insofar as the naming of data variables is concerned can be summarized at this point. Use short names. Make them unique within the first two characters. Name those used most often, first. Keep the total number of names used to the minimum. Make balancing trade-offs between the use of simple variables vs. arrays. Avoid conditional expressions that use names that have never been added to the tables. For those who like to CHAIN, be cautious of passing names along; those passed on will be at the top of the tables in the program that inherits them. Focus optimizing efforts on names used in long running loops and those used in interacting with the outside world. (An operator will enjoy the fastest keyboard response you can provide, for example.)

One more caution is needed. Notwithstanding my enthusiasm for providing objective guidelines, TU as used here must still be seen as a relative yardstick. The interpreter's need to move groups of bytes from memory into the CPU can skew precise timing comparisons of different machines (viz, 8-bit buses vs. 16-bits). Also, a minute difference between successive tests on a given machine can occur because no ORG is done to ensure that the tables align on even-word boundaries.

And here's a plug for Chapter 15: One of the handy tools included there is called VLIST. It provides a quick list of what names are actually in the tables, in the order in which they are stored, at any selected point in the execution of a running program.

What is in the variable-names tables: The names themselves, of course. Immediately following each name, space is allocated to contain its data. For numeric data, this is the space used to hold actual values. For string data, this space contains the address of where each string is physically located. (Again, Chapter 4 documents where that is.)

A name entry in a table is at least four bytes. Bytes 2 and 3 are the first two characters of your name, in upper case ASCII. If a name has only a single letter, byte 3 is zero. If a name is for a user defined function, byte 2 is equal to the ASCII chart value of that name's first letter, plus 128 (decimal).

The fourth byte of a name is a VLI (Variable Length Indicator). This is a count of the number of characters in that name, less two. That is, it is the number of bytes that follow the VLI. For one or two character names, the VLI is zero. Each of the bytes that follow the VLI--the remainder of the name, if any --are equal to their ASCII character equivalents, plus 128. (Note that adding 128 in decimal is the same thing as turning on the high order bit in a byte in machine languages.)

The single byte that precedes each name entry is a number. For numeric variables the number is 2, 4, or 8, which is the number of bytes that are needed to store an integer, or a single or double precision value, respectively. The number that precedes string names is 3, the number of bytes needed to hold a 1-byte VLI of the string itself, and the 2-byte address of where the string is actually located.

How the data-type indicator is determined: If a name in a line of code includes a type appendage, like Z$, the dollar sign is translated to 3, for example. If a name has no appendage, its "default type" comes from a string of 26 codes being maintained in the interpreter's own working storage. The first letter of a name is used as an offset--as in ASC(name)-64--to get the current default for all names that start with that same first letter.

The inference in the paragraph above needs to be stated. Note

that data-type appendages are translated. And dropped. They are not stored as a name-character. So, XY$ is really the same length in the tables as XY. In both cases the type-code will precede the name (assuming an earlier DEFSTR X). Knowing this, by the way, performance debates should avoid arguing about whether or not to use self-typing variable names. There is a performance advantage to names that have appendages. It is so slight--something like .02 TU--it ought to be ignored. (My programs use DEF-type, and no name-appendages to save wearing out my little pinky on the typewriter shift key. My performance matters, too, not just the machine's.)

Now we can read between the lines in the language manuals. Read: "...BASIC's default data type is for single precision numeric variables..." means that, when the interpreter is first loaded into memory, all 26 of the codes in the list of defaults are initially set to 4. And: When we do DEFINT C-L the codes in list positions 3 through 12 are changed to a 2, for example.

Further: "A type declaration appendage takes precedence over a default type..." means, really, if an appendage is included, the defaults-list is not looked at. 'Lo and behold, with a DEFDBL X currently in force, X and X$ can be seen as referring to different variables. Conversely, after DEFSTR X, X$ and X mean the same thing.

> Note: Although it is not abundantly clear in the manuals, nothing "happens" to the data stored in variables whose type gets redefined. If A was originally numeric, and contained a number, DEFSTR A can be done so that a string can be stored by that name. Now, DEFDBL A will again point to the number, and A$ will point to the string. Care must be taken in all cases involving numerics. If A was originally for an integer that number can only be referenced by A%, or by doing a subsequent DEFINT A, for example.

Another stray bullet is needed here about data typing:

> Compilers and interpreters work differently. Whether or not variables should be "self-typing", and how and when DEF-type statements should be used are matters that require serious thought for programs that may be run in either environment. The conventions used in my shop address this fully; Chapters 10 and 12 both have comments that fully explore this issue.

Inside the tables: They are alike. A 1-byte data-type code, the first two bytes of the name, a VLI-byte, and the rest of the name, if it has more than two characters. For simple variables the rest is simple. The data itself comes next, i.e., in the next 2, 3, 4, or 8 bytes. Arrays have several more bytes of overhead that follow the name, ahead of the area allocated to contain the data.

The next two bytes after an array's name contain a total-bytes count needed for the data area. (This value, plus 2, added to this counter's own address is the address of the data-type byte that begins the name entry of the next array.) The very next byte after this VLI tells how many dimensions the array has: DIM GX(7,4) has 2 dimensions, one for 7, and one for 4.

The dimensions-indicator byte is followed by 2-byte counters, each counter telling how many elements are in each dimension. In the above example, the first counter would be 5, and the next one would be 8. PS: The counters are in reverse order, as compared to how they are stated in the DIM. Notice also, if you did not specify an OPTION BASE 1, the zero-element is included in these counters. Either way, with or without a global OPTION BASE statement, the dimensions indicator is a literal count.

Now we can deduce the "computing" the interpreter does when it searches for variables. And appreciate why it does take some time, sometimes.

Beginning at the beginning, to locate the next simple variable, add the first one's data-type (2, 3, 4, or 8). Add 2 more to get the VLI, and add it. The result, plus 1 (the width of the VLI) points to the data-type indicator of the next variable.

Finding the next array variable starts out the same way: The data-type, plus 2, plus the name's VLI. Now, adding the value in the next two bytes--the array's VLI--plus 2, points to the data-type indicator of the next array variable.

Whee. The manuals are now elucidated. An array declaration could theoretically have 255 dimensions, the capacity of the 1-byte dimensions-indicator. Some manuals actually say this. Others are more accurate and point out that it is impossible to construct a line in a program that would specify that many dimensions. (By the way, a zero-dimensions array cannot be

specified either. Nor would it serve any logical purpose.)

Each dimension could, in theory, indicate the maximum capacity of the 2-byte counters. Some manuals specify a maximum. Some don't. It probably does not matter anyway. Real programs are bound to run out of memory long before the counters run out of bits. With only 60k or so of usable program memory, a one-line program would run out of memory trying to declare an integer array larger than 30,000 elements. Or thereabouts.

With 30,000 as the problematical maximum for integer arrays, it is one-fourth that for double precision numerics, or, about 7,500 elements of 8 bytes each (vs. 2-byte integers). Single precision numeric arrays, with 4-byte elements, will run out of memory twice as fast as integers; at about 15,000. Strings require three bytes of overhead per element, so if only one data character was stored per element, their limit is also about 15,000. If no data are actually loaded into a string array, as many as 20,000 elements might be definable. Once more, these are maximums. They are naturally less depending on the space used to contain your program itself, space needed for file buffers, and so on.

Getting back on the main road after that side-track, the mapping of data elements in arrays needs to be documented. Those heavily into math may do intellectual tricks using terms like vectors and matrices. Others--including yours truly--may draw little crude charts to figure out complicated FOR/NEXT indexes, subscripts, offsets, or simply: Pointers into arrays. However we do it, it is for the purpose of devising the logic for conventional access to individual data elements. For the unconventional, for those instances where it is prudent to use VARPTR and PEEK and POKE tricks, it is necessary to know how the elements are actually stacked up in array storage areas.

The overall design of some programs also demands an awareness of how memory is allocated for arrays, an awareness that is hard to glean in a casual reading of most BASIC manuals. An ERR = 7, "out of memory", can be embarrassing to most veteran programmers; the flushed face of an amateur merely indicates incomplete mastery of his burgeoning skills.

Each named array is a subtable. The subtable begins right after the name's header-string of bytes. The individual elements are placed contiguously--one right after another--with nothing physically separating them. This is possible

because all elements of a given array are of exactly the same length. Yes, even for string data. Not to beat a dead horse, but remember that in a string-array, each element is exactly 3 bytes, the VLI-byte followed by a 2-byte address of where each (variable length) string is itself actually located in memory.

For the simplest example, like DIM AX(5), the cost in the table is a multiple of the bytes needed for whatever data type A is, times the number of elements. If A-names are integers, and no OPTION BASE 1 is in force, the table for this array would be exactly 12 bytes. Two bytes for each element, position 0 through position 5.

Continuing with this example, an expression referring to AX(0) accesses the value stored in the first pair of bytes. AX(1) is a reference to the next pair of bytes. AX(2) refers to the next two, and so on.

Another momentary aside: When you paraphrase a manual to say that you get a free DIM out of an expression like AX(3) = 7, with no explicit DIM for AX having been done beforehand, space is allocated for a full 10 or 11 elements--for each dimension-depending on a stated OPTION BASE, or, the default BASE 0 that is established when the interpreter is initially loaded.

Multidimensioned arrays are mapped nearly as simplistic as single-wide tables. Each successive dimension has its own subtable of contiguous elements, and the subtables are located one after another in the same order as the subscripts are named in an expression. The size of each subtable can be computed by multiplying all dimensions, then by multiplying that result by the number of bytes needed for each element.

Using an example to clarify this, assume OPTION BASE 1 and DIM X$(2,4,6). The first subtable is 2*4*6, for 48 bytes, times an elementary length requirement of 3 bytes per, for a total of 144 bytes. The second subtable is also 144 bytes. And so is the third. For a total allocation of 3*144, or 432 bytes for the entire array.

Compare this with OPTION BASE 1 and DIM X$(2),Y$(4),Z$(6). The X$ table-allocation is 3*2 = 6 bytes. Table Y$ is 3*4 = 12 and Z$ is 3*6 = 18. Which is 36 bytes total. Which is quite a bit less than the 432 bytes in the preceding example. Most of the time, for small arrays, memory consumption is a weak topic of

debate. Large masses of data, on the other hand, tend to dictate using linear, single-dimensioned, individually named tables rather than multidimensional arrays. Caveat and finito.

A few parting remarks are needed to round out this subject in full. The start up section of a program should declare all names that are used anywhere in the program's fabric. Name all simple variables first, then arrays. (Remember that all arrays must be moved downward whenever a new simple variable is added; large arrays mean large blocks of memory must be relocated.)

Develop habits that will not contradict QuickBASIC. To wit: DIM all arrays. Even the little ones. GW lets you "default" small arrays with eleven or fewer elements. QuickBASIC insists on the use of DIM in all cases.

User defined functions is another semantical ambiguity between these two languages. In GW you can define a function that names an array before the array is declared (dimensioned). Not so in the (dumber) compiler. So, name simple variables, then user defined functions that use only simple variables, then arrays, then user defined functions that have to reference arrays. (And tolerate the burden that a user defined function declared after arrays have been named will cause the array blocks to be shifted downward in memory.)

Finishing this chapter, after laboring through my laborious narrative, hopefully you will be rewarded by writing faster running programs, and when forced to, by being able to put five pounds of sand in the proverbial four pound sack. In either event, my reward is in sharing what I have learned over a long period of time. And some of it took me a long time to learn. Whether that was because I am dense or the manuals are obtuse can certainly be argued. My contention remains:

> To be able to write efficient programs requires a thorough understanding of how the interpreter works.

Chapter 4 = STRINGS

It is certainly possible to write acceptable programs with no concerns whatsoever about how the interpreter works. Much of this book is devoted to the theme that you can achieve superior performance, however, by using techniques that are the most efficient, based on a total awareness of how the interpreter does work.

Sometimes the net difference between choices made by a novice vs. an old hand is hard to measure. Sometimes the difference is even as much as a few minutes, but no one cares. Two or three minutes longer than need be, for a program that is used only once a month, with a typical overall run time of only ten minutes is not apt to get any programmer fired. On the other hand, the essence of this chapter is critical for professional survival.

The theme here is not only on how well a program works. Some may run well enough, initially, to be acceptable to the one paying for it. Some may run for weeks, or even months, before a programming faux pas becomes evident.

The risk: A program is written that is one of several making up a total application set. It is discovered to be intolerable some time after the application is installed and its owners have become dependent upon it. A fix now is going to be expensive. For someone.

This type of risk is always potential. Of course. When the fault is your own, the cost for repairing it becomes yours also, usually. Anyone that has written more than a few sizable programs has learned to live with the aggravation caused by an occasional bug. A bug inside a program, that is. Design bugs, on the other hand--those that permeate the overall scheme of how a program was mapped-out in the first place--can be much more than aggravating. They can mean misery. Seldom is it fun to have to completely rewrite any program because its design was based on invalid assumptions.

How strings are managed by the interpreter is one area of knowledge sorely needed to preclude making serious design mistakes--knowledge that in many cases can be gotten only at U.H.K. Tuition rates at the University of Hard Knocks can be very expensive.

Suffer my continuing criticism that many manuals are obscure. In one case cited below, so erroneous even, that its advice could cause you to make design decisions that you will later regret. The purpose of this chapter is to cut learning costs.

For some this may be only a refresher course. Others may find even more of value. The cost of a few minutes reading time is bound to be a cheaper lesson, for anyone, than having to learn from costly mistakes.

As you read what follows maintain a mental image of a simple memory map. An interpreted BASIC program is a continuous string of bytes, a block of text if you will, that is much the same as it is stored in a file on disk. A file created by SAVE without the "A" option. What the bytes of that text actually look like is unimportant at the moment. (Chapter 2 documents all of that.)

When you LOAD a program--or create one while in editing mode--the text of the program lines is stored in memory as a "block of bytes". For this narrative, see this as block number two, of five. These blocks are soft subdivisions of a single chunk of memory that a GW-BASIC program runs in. The overall size of this chunk cannot be larger than 64kb.

Block-1 is a working-storage area for the interpreter's own use. Its size can be varied within certain limits by use of the "slash options" when the interpreter is first loaded. From then on, the size of this block remains fixed.

Block-2 is your program. Its size is initially similar to its "file size" as it is stored in a disk file. This size remains constant while a program is running. While you are editing, this block increases or decreases in length as you add or delete program lines, or change anything that alters the physical length of a given line.

Block-3 is where your variables are stored after you do a RUN. This block is conceptually a table of names; each name is immediately followed by what is "in" that variable at a given moment. Chapter 3 describes this block, fully. For the subject at hand, remember that what is actually in string variables, in this block, are the addresses of where strings are, not the strings themselves.

Block-4 is generally called "string space". This is where

all "strings" are actually kept. This is the block that this chapter focuses on. It is that heap of bytes that is bounded on the north by the bottom of block-3, and on the south by the beginning of block-5.

Block-5 is last. Its size is fixed while running. Most manuals allude to this as the interpreter's "stack". Its size can be altered by use of CLEAR. The bottom of this block can be considered as the end-of-memory for programs that use conventional techniques. This soft boundary is normally set for the maximum of 64k bytes, but it can be set to be less, by use of a slash-M option when the interpreter is first loaded.

Bearing this simplified map in mind, remember that blocks-1, 2, and 5 are static in size while a program is running. Block-3 grows downward. The space in block-4 is used from the bottom up. When the bottom of three runs into the top of four, or vice versa, hum the jingle in the beer commercial:

"When you're out of FRE(""), you're out of space...."

When this happens, here is what the interpreter does: It runs through your stack of variables (in block-3) looking for those that are for strings whose addresses point into block-4. As each one is encountered, its string is moved, if need be, so as to wind up with all currently in-use strings packed end-to-end at the bottom of block-4. When it gets done, if some space was freed up as a consequence of overlaying strings no longer being used, the program keeps right on running. Or walking. The execution pace can slow to a crawl even. Read on.

If only a little bit of garbage gets discarded you may not even be aware that it happened. Sometimes it takes long enough for you to hum the tune suggested above. In some cases it can take long enough for you to go to the store and replenish your beer supply.

No matter how long the garbage collection takes, in the event you really are out of memory, and you get the dreaded ERR = 7, a trip to town might be a good idea anyway: To look for a new job.

If my sense of levity tries your patience, remember the manual that said--in the staid manner of all such literature--the time

required for repacking strings can take "...a minute to a minute and a half". Or something like that. If my quote is imprecise, my remembrance of what it said is accurate.

Where that range of 60 to 90 seconds came from is a complete mystery to me. On an older, slower micro it can be as little as a mere few milliseconds. In contemporary machines (8088), running at 10 MHz, it can take half an hour. Or more.

Melodramatic? Not at all. Wrote a POS (Point Of Sale) program for a shoe store. Held the SKU (Stock Keeping Unit) codes in a single-dimensioned string array. Four thousand codes, mostly 6-bytes each. About 36kb used for this. (Prices and inventory balances were kept on disk.) Total program size was about 56kb. No sweat; ran beautifully on Wednesday. And Thursday. Big sale started Friday. Heavy traffic all day. Boom. About midday, with nine people standing in line, the "cash register" goes out to lunch. And it stayed gone for over twenty minutes.

Fact or fiction? Try this:

```
DIM X$(3999)                        'space allocation
FOR I = 3999 TO 0 STEP-1            'decreasing loop
 X$(I) = STR$(I)                    'phony look-up code
NEXT                                'fill whole table
PRINT TIMER                         'mark start time
PRINT FRE("")                       'provoke a clean up
PRINT TIMER                         'how long to do?
```

Continuing with empirical research, we can deduce that if the codes were stored in ascending sequence, instead of backwards, the clean up would only take 13 or 14 minutes. That would be better. Maybe only half of your customers will get mad at you. Far better still, don't make anyone mad.

There are several tricks and techniques that can be used to stave off lengthy time-outs, and a couple of others that will enable you to warn an operator of an impending pregnant pause. Those suggestions are at the tail end of this chapter. That is where they deserve to be. They ought to be used only when doctoring programs that were malformed in the first place.

Only one suggestion needs to be heeded to ensure that no well written program will ever go into an uninterruptible time-out: Once a program gets rolling, make sure that the size of blocks three and four remain static. Static enough, at least, that it

can be predicted that their boundaries will never collide.

The coding techniques that can be used to freeze dynamic storage areas are not exotic, nor even contrary to what some would call "standard programming practices". To be able to predict, and therefore to preclude unnecessary "garbage" in string space is not difficult either. But it does require an awareness of what causes fragmentation in the first place.

Some of this knowledge can be intuitively surmised from reading programming manuals. Some is so subtle that it can be gleaned only by extensive probing. And some of us are more adept at reading between the lines than are others. What follows makes no arbitrary discrimination. It is the sum of it all that is necessary to keep in mind while programming. In all programs.

If a statement in a program says PRINT "Hello", there is no impact on any memory utilization. The interpretive routine that executes PRINT gets its output directly from the quoted literal that is imbedded in the fabric of your program.

In the case: X$ = "Hello", may cause a small impact. If this is the first time X$ has been encountered, that name must be added to the tables in working storage. In my simple map above, the size of block-3 must be lengthened to hold the new name (4 bytes), plus the "overhead" for the interpreter to know how long this string is, and that it is located up inside your program. (Some manuals say the overhead for strings is four bytes. I count three; the length-byte, plus the address pair.) The bottom boundary of block-3 just moved downward 7 bytes.

Contrast this with: X$ = STR$(2*400). If the variable's name is already in working storage, the size of block-3 remains the same, but the interpreter needs to construct a 4-byte ASCII string: A space, an 8, and two zeros. When doing so it uses the next four bytes just above the current roof of block-4, the area of memory that thus far, hopefully, is unused for anything else. After the requested string is generated, the variable's name-entry is updated to a length of 4, and the address-bytes are overlaid with the address of where this string is actually at down in block-4. And the interpreter's work registers are updated to show that the top of block-4 has just moved upward four bytes.

Do it again. X$ = STR$(2*400). Our manuals are sorely remiss

on this: Another 4 bytes of memory are used to hold this newly manufactured string, even though the target's space requirement is exactly the same as it was. One space, an 8, and 2 zeros. And, its name-entry in the simple variables table is now updated to point to a "new" address. The result is a hole. The four bytes used for the previous string are now, unfortunately, an unaddressed free-space fragment. They are garbage.

Do it differently: LSET X$ = STR$(2*400). Take advantage of an important opportunity implied in the manual's description of how LSET and RSET work. They reuse string storage, if it has been previously allocated.

FIELD, LSET, and RSET were added to BASIC in the early days of random access files. (Read: Disks.) There is still a trend in the literature to imply that LSET and RSET only work with "fielded variables". In fact, they work quite well with any string variable.

When you FIELD #1,7 AS X$, the variable's name is added to the tables in block-3 on exactly the same basis as if you did X$ = "Georgia". Or, X$ = SPACE$(7). The only difference is, where the string itself is.

Using these same examples: The fielded X$ is up in block-1, in the interpreter's working storage, in an area set aside as a buffer for your file #1. As described earlier, "Georgia" is up in block-2, a literal in your program. And the 7-spaces were manufactured on the fly, down in block-4 in the so called "string space".

Now we read between the lines: If you do a LET to a variable that was previously fielded--instead of an LSET or RSET-- "...the logical association of that data to a file is lost." Because the string that is now pointed to is down in block-4, not, as it was, up into the file's read/write buffer.

Before stumbling on we should know of a quirk that is not even implied between the covers of the manuals. Bilingual programs that have to work in either interpreted or compiled mode have to contend with this:

> When a data file is opened--which must be done before its buffer can be defined with fielded-variables--its buffer's space-allocation (and its contents) remains unchanged after

the file is closed. That same work area can be reused for storing any string with LSET, RSET, or MID$. If the same file is opened again--the same file number, not necessarily the same file specification--previous field statements are still in affect, if no redefinition of those variables was done in the interim.

Here's the catch: The above paragraph is totally true when using the interpreter. It is absolutely false when using the compiler. Don't bother to look. It is equally true that, none of this is mentioned anywhere in either of the BASIC manuals. (PS: OPEN initializes all fielded variables as hex-zero bytes, not as spaces--another gotcha not in the manuals.)

When variables are named in a field statement, if they had been used elsewhere previously, the strings they addressed before are abandoned: More junk for the garbage collector.

Mark this point before racing on: The growth of block-4 can be contained by pre-allocating working storage for strings, and then, by making maximum reuse of the space thus set aside. Do it like this: During the start-up of a program, declare fixed-length work strings. For example:

```
W1$ = CHR$(0)            'for single byte operations
W2$ = MKI$(0)            'for pairs of bytes
W4$ = MKS$(0)            'for 4-byte words
W8$ = MKD$(0)            'for 8-bytes at a time
WB$ = SPACE$(80)         'for a whole (print) buffer
```

Now, as coding proceeds, use LSET, RSET, and MID$ to make continuous reuse of these worker-variables. (Not only will this constrain the growth of block-4, overall performance of your program will usually be better).

String "functions" were used above to force the interpreter to pre-allocate string workspace. Compare this technique with using W4$ = "...." instead. Remember, when a quoted literal is first declared, the variable will point up into your program. Later, if you do an LSET to a variable that points to program text, a string of bytes equivalent to the length of that quoted string must be allocated at that time, down in string space, before the LSET can be accomplished.

In any event, ignore the ill-formed advice cluttering up the manuals. Yes, X$ = "Hello"+"" will force a string to be placed immediately down in "free" memory. (Because of concatenation; regardless of the fact that, really, nothing is being tacked onto the word Hello.) This seemingly innocuous little trick may also trigger a garbage clean up. Even if there is not enough room left for only the null, which needs no space at all.

Some of the above can be learned by reading manuals closely. By reading what they say literally, and by inferring what they almost say. But, more must be known before you can program so as to not get any nasty surprises. Nasty? Because you cannot predict.... If you cannot predict, you cannot promise. If you cannot promise, do you expect to get paid? Compare:

```
100 PRINT CHR$(34);"hello";CHR$(34)

200 PRINT CHR$(34)+"goodbye"+CHR$(34)
```

Knowing the ASCII character for code 34, even elementary students can tell you that the two lines above will print a greeting and a salutation bracketed by quotation marks. Some seniors might notice that line 200 uses concatenation.

The very astute might even spot what older veterans have learned the hard way (maybe). Line 200 just created 9-bytes of garbage. The interpreter does all concatenation down in string-space memory. And, it just moved the top boundary of block-4, 9-bytes closer to colliding with the bottom of the variables storage area. The program is now 9 bytes closer to the moment when a time-out must be done if a collision does occur. Unnoticed concatenation can cause more traffic jams than unseen ice on a freeway.

PS: Line 200 runs slower than 100; "goodbye" had to be moved (copied) from one place to another before it could be acted upon. In line 100, "hello" can be printed right from where it is, in program text-space. This difference is small, for small things, but can be very significant for lengthy output.

The overall concept of variable length strings is beautiful. To be able to concatenate--to be able to hook two strings together, end-to-end--is even better. And it is so easy to do. So much so that it is enough, almost, to make one want to think that BASIC is a higher level, "high level language",

than COBOL. Little that goes on in a COBOL program is hidden from the programmer, however. He does not have to worry about running out of memory unexpectedly. In COBOL, all storage areas must be explicitly allocated ahead of time by the programmer himself. In BASIC we can let the interpreter take over the burden of managing space for us. If we do rely on that, as we are coding, we must maintain a mental tally of the litter we are leaving behind.

The following is another example of string concatenation:

```
X$ = "Hello"
Y$ = "and"
Z$ = "goodbye"
M$ = X$+" "+Y$+" "+Z$+"."
```

What is the memory fragmentation cost in this case? 42 bytes. To determine this, do a PRINT FRE(0) just before this segment of code, and again just after. The difference in the two numbers printed will be 60 bytes. The resultant length of M$ is 18. And 60 minus 18 is 42. To duplicate this technique of measuring, know that all of the variables used (X$, Y$, Z$, and M$) must have been previously encountered in the program so that their table-overhead has already been accounted for.

If you prefer bookkeeping over programming, so be it. My preference is to not have to bother with keeping track of how many bytes will wind up in the bit bucket. Besides being a chore, a small accounting error could exact a large penalty.

One obvious alternative is to avoid doing concatenation. That is sage advice. Like all such advice, it must be tempered with realism. For a small. quick-out program that can afford to burn the bytes because there are so many, and the run will finish before the memory runs out, no problem. For full scale, real applications that have to run for hours and hours, if we borrow that concept from COBOL--of pre-allocating storage--then we need not be bookkeepers. Or be paranoid.

Using the earlier example of WB$ as a general purpose worker, here is one alternative to doing concatenation.

Given:

```
X$ = "Hello" : Y$ = "and" : Z$ = "goodbye" : WB$ = SPACE$(80)
```

then

```
  LSET WB$ = X$ : MID$(WB$,7) = Y$ : MID$(WB$,11) = Z$
```

then

```
  PRINT LEFT$(WB$,17);
```

will produce a "concatenated" message with absolutely no cost. No cost in the sense of causing any memory fragmentation. The trade off, obviously, is that the cost is now one of effort on your part. Calculating, exactly, the arguments to use in the MID$ and LEFT$ statements is tedious. Pay now, or pay later.

One more incidental must be considered before assuming that there will be enough "free space". The above described what to do: Make maximum reuse of already allocated string variables, and avoid doing concatenation. The do-not-do list of rules needs to be longer because, incidentally, the interpreter's appetite for free bytes is voracious.

Here is an example of a do-not-do. It is not a very useful algorithm. It is merely a method of conveying an important insight: Choosing a technique for how to do something must be done with care.

```
   200 IF RIGHT$(TIME$,2)<>"00" THEN 200
   210    PRINT LEFT$(TIME$,5)
```

When the interpreter bumps into the word TIME$ in line 200 it constructs an 8-byte ASCII string, like "07:14:52". It does it in string-space, immediately above the current top-of-block boundary, of block-4. It then subtracts 8 from that block's beginning address. And it does it again, and again, and again, until the seconds roll past 59 and become zero. Then it does it one more time, to get the hours and minutes for line 210.

The above is not only bad technique, it may not work at all. Line 210 may never be gotten to. On a machine running at 8 MHz, over 24,000 bytes would be burned every 10 seconds.

If the looping burns more bytes than available, at that point in the loop, a time-out will occur, to free-up "free space". And to do so, remember, it has to run through your variables

and pack your strings, regardless of the care you have taken to preclude fragmentation. And while it is doing this chore, necessary or not, the execution of your timing loop has been suspended. And while the interpreter is off cleaning up its own mess, time rolls on. What are the odds that "00" was passed by in the interim? You could be trapped in this loop for a long time. Forever, even. Or at least until a lucky coincidence in time lets the interpreter conclude your test before it runs out of memory again.

So: Wholesale concatenation is a no-no. And so is using substring functions--or implicit concatenation--whenever it will cause the interpreter to construct a full length ASCII string of text, from which to extract the requested substring.

Like: PRINT STR$(789) has no impact on string space. This construction is done up in block-1, in the interpreter's own work area. Oddly enough, PRINT MID$(STR$(789),2) -- to keep from getting an unwanted leading-space--will cause all four ASCII-text characters to be strung together, down in your "free" string-space first, before the PRINT interpretive is directed to print the three you want from there. And, at a cost of pushing the top of block-4 closer to an unexpected time out, all because of four, very temporary bytes.

So, don't do it that way. Do LSET W4$ = STR$(789) then do a PRINT MID$(W4$,2). Which is also an alternative technique, albeit a not very elegant one, that would work successfully for the timing-loop above. Viz: LSET WB$ = TIME$, then the test, IF MID$(WB$,7,2)<>"00".... In this way you decide where the test-string will be constructed in memory. What is often called "free space" may be free to the interpreter, but it can be costly to you.

All of which brings us to the decision making point: Before a program is written. It's called designing. What must be done? Where can you put it? How big will it be? Roll these three answers around, and around, and determine if there will be enough memory. Not just to hold it all, but will it be enough until the job is done? The answer to this last question should determine the discipline to be used when coding. To use memory conserving techniques, or, "... damn the torpedoes, full speed ahead." Which may equate to not much speed at all if you SWAG wrong about the memory needed. (SWAG is a technical term often used in software engineering: Scientific Wild-Assed Guess.)

A large string-array is slow to clean up. Each element is not unlike having an equal number of individual string-variables, when we use FRE(""), Which can be used to deliberately provoke the repacking of string-space. Which is a way to be able to warn the operator that your program is about to pause for the cause. Which is better than letting the interpreter decide when it needs to be done, with no warning to anyone.

Continuing with my promise at the beginning of this chapter: You could do something like A = FRE("") at arbitrarily fixed intervals. Maybe at the start of a menu-selected task, for example. As a better alternative, in some programs, force a garbage clean up, upon completion of each major task.

An alternative method of knowing when to signal for the dump truck, to get rid of your garbage, is to monitor when it's apt to show up unbidden. Pick an arbitrary waste figure, like 4,000 bytes, and periodically test to see if FRE(0) has fallen below this threshold. When it does, jump to your subroutine that warns somebody it's time for the machine to take a break.

A restart is another method that can be used. Sometimes. It is not always easy to do, but if your union boss is sensitive about unscheduled breaks....

A simple RUN resets everything, and no garbage clean up is necessary (or done). If a start-up involves opening files, or includes soliciting one-time input from an operator, this idea may not be too attractive. In a few cases CLEAR may be used to avoid going all the way back to the procedural beginning. Doing either of these, or anything similar, may also entail having a mechanism for remembering what was in some variables, to be able to keep on running as if nothing untoward had happened.

The "anything similar" hinted at above could involve moving large string arrays outside of the program altogether. This might be good salve for a sore spot, where it is the time that clean-up takes, rather than the frequency with which it must be done. One must consider where the strings for such an array come from in the first place, of course. And whether or not they could be kept in a file instead of an array. VDISK can sometimes be used as a viable alternative, for example, with only a nominal impact on overall performance.

A middle of the road answer for problems involving large string-arrays must be mentioned, for the benefit of those who just might not have already discovered it. Short strings can be stored in numeric-arrays. Although they too take up space, they do not slow down the garbage collector. For example:

```
DIM A(2000)                 'DEFDBL A is in affect
X$ = "hello   "             'any 8-bytes
A(999) = CVD(X$)            'ERR = 5 if LEN(X$) < 8
LSET X$ = MKD$(A(999))      'looks just like it did before
```

Some of the solutions suggested above straddle that fine line between "good programming practices" and the unorthodox. (Chapter 10 gives my arguments, pro and con, for making professional, ethical compromises.) A well mapped-out, well coded program, or set of programs, should not have to resort to makeshifts like having to announce the need to time-out.

If my recommendations seem to you to be, to develop coding habits slightly different than what might be considered as "normal BASIC", appreciate from whence they come. My ambition is to pass along experience garnered from a lot of programs. Some were better than others. The best, invariably, were those where I maintained a total cognizance of how the interpreter works.

Thus my continuing contention: The same will hold true for you. Your best programs will be those that are coded in full consideration of how the interpreter works. Bearing in mind, also, the vagaries of the compilers, programs efficiently coded for the interpreter will also behave efficiently if they are later put into production as a compiled package.

Chapter 5 = NUMBERS

My most frequently used BASIC manual devotes three whole pages to numeric constants, and how they are stored in memory, using highfalutin terms like floating point, exponential notation, and mantissa.

Contrast this with the forty-odd pages devoted to "graphics", and ten or so more to playing music. Some call me cynical. They should have to write data processing programs on today's computers using software more akin to playing games than doing accounting.

All of that pompous jargon that is used in those few pages about numbers is meaningful, no doubt, for those figuring the number of atoms in a billiard ball. Only a word or two can be found, however, for those of us doing the mundane with money. For money, or for ourselves.

A small part of this chapter will deal with small numbers; the greater part, with the greater problem of dollars and cents. Which also embraces great big totals, like for Yen. Knowing where most of the chips in my machine came from, it would seem those folks, at least, would critique our manuals to be sure that we can compute their export payments precisely.

Textbooks about computers invariably begin by talking about binary numbers. We can skip that tedium here. Only a short review of the elementary is needed to set the stage for the real bit busters.

> The bits in a byte--all eight of them--are still just that, no matter how that byte is offered up for the benefit of human consumption.
>
> Agreed: A pattern of 0100 0001 is the traditional way to show the contents of a byte in binary. In this case, if shown as a hexadecimal value, it would be 41. If shown as a decimal number it would be 65, or if treated as an ASCII printable character, it is the capital letter A.
>
> In all events, it is still an 8-bit byte. How it is to be exhibited is up to the person wishing to communicate with another human. The bits in the bytes themselves, inside the machine, are still just that. Some are on and some are off.

Whatever that pattern is, it remains the same, no matter who is eyeballing it.

Agreed: PRINT "A" and PRINT CHR$(65) and PRINT CHR$(&H41) will all print the letter A. If X$ = "A", and Y$ = CHR$(65), and Z$ = (&H41), then ASC(A$), and ASC(Y$), and ASC(Z$) will all, also, equal 65, or 41H, and so on.

At least we no longer have to fool with octal, which we used a lot on the old PDP-8 machines. Hexadecimal is still in vogue, however. Perhaps because it is a shorter form of notation than octal. And monitors can display alpha characters as well as numbers. Octal had the advantage back when all we had were digital displays; the digits 0-7 could be used to display very large values, using much less space than would be needed to show the same thing in decimal notation.

Now that we have big screen displays, and seldom need to do memory dumps to paper, it is unfortunate that even hexadecimal persists. Granted, some people may be impressed by those who can speak "hexadecimal".

The folks most responsible for perpetuating the hexadecimal hex are probably those that contribute to the development of assemblers; some of them really do seem to like it. (We mortals grew up learning to count on our fingers. Base 10, not 16.)

Pick up any BASIC manual that has DEF SEG in it. The example shown is always: DEF SEG = &HB800, the first address in screen memory (CGA). Why is hexadecimal used and not decimal? They could show us: DEF SEG = -18432 (or DEF SEG = 47104! because a negative address does look a bit perverse).

There is a very small advantage to using in-line hexadecimal literals--&HB800 is imbedded in a "tokenized program" as two bytes, whereas 47104 uses four--but this same advantage can be had by using the negative complement of 65536 (i.e., -18432).

Unfortunately, there is no "intrinsic function" in BASIC with which to obtain the full range of positive numbers that may be stored in a 16-bit word.

Here are two "user defined functions" that can be useful for overcoming the clumsiness of negative addresses.

```
  Given:    DEFSNG B:DEFSTR M      'define data types
            M = MKI$(-18432)       'start of CGA memory

   Then:    DEF FNB = ASC(MID$(M,2))*256+ASC(M)

     Or:    DEF FNB = CVI(M)-65536*(CVI(M)<0)

    Now:    PRINT FNB produces 47104, rather than -18432
```

Performance is the same for either of these; either will work with the interpreter, but only the second one can be compiled. (The compiler will permit the first one, but gives an overflow error at run time. It is extremely hard to anticipate what the compiler will do with your arithmetic expressions. It abhors multiplying and dividing more than most fourth-graders.)

Here is another one: DEF FNB = LOC(1)-65536*(LOC(1)<0). This is useful to GET 1,FNB-1 because you can GET and PUT way beyond 32767, but the LOC function returns a negative file pointer if you do. (Chapter 8 has loads of hard-won info about files.)

So, 2-bytes, 16-bits, can represent 65,536 unique bit patterns. If one bit is used as a sign-bit, we cut the capacity in half, but now both negative and positive numbers can be stored in the same space; the range of numbers possible is then from a low of -32768 to a high of +32767. This is what BASIC calls integers.

Agreed: &HB800 and -18432 are two different ways to "print" the same thing. In modern micros--8080, Z80, 8086, 8088, 80286, 80386, 80486--numbers are stored (in memory) in 2-byte words, with the two bytes in reverse order if we are reading from left to right.

If we assign -18432 to an integer variable, then find those two bytes in memory using DEBUG--which displays memory contents in hexadecimal--they will be seen as 00B8. Which is the same as if we had used &HB800 as a literal. See the similarity, and that the two bytes are "reversed".

Do not let the hyperbole in the manuals confuse things. It does not matter to the micro how the bits in a word get set. See this:

```
      DEFINT C-L:DEFSTR M-Z          'define data types
      X = MKI$(-18432)               'memory = 00B8
```

```
I = ASC(X)                      'I = 0
C = ASC(MID$(X,2))              'C = 184
PRINT HEX$(I)                   'prints 0
PRINT HEX$(C)                   'prints B8
PRINT C*256+I                   'prints 47104
PRINT C*256+I-65536             'prints -18432
PRINT HEX$(C*256+I-65536)       'prints B800
```

Suppose you want to print a character on the screen then move the cursor so that it is superimposed over that character.

Here is one way to do it, slowly...

```
LOCATE 10,30:PRINT "?":LOCATE 10,30
```

Here is a faster way...

```
LOCATE 10,30:PRINT "?";CHR$(29);  '29 is a backspace code
```

Here is an even faster way ...

```
LOCATE 10,30:PRINT MKI$(7487);    'CVI("?"+CHR$(29))=7487
```

See how the use of MKI$ in the above can output two bytes to the tube (conceptually, a serial device). It can be done to a sequential file too. So why all the bull about CVI, CVS, CVD, MKI$, MKS$, and MKD$ being for the benefit of random access files? Who do they think they are kidding: "BASIC stores data in random files in a compacted form." What is in files is up to the programmer that generates them. (Chapter 8's theme.)

"Convert Variable Integer" was the original mnemonic intent of CVI. MKI was a clumsy acronym for "Make Integer". In neither case does anything get "converted" or "made". This is true for CVS and MKS$, and CVD and MKD$, also. All of these functions are merely useful for telling BASIC how you want to store your own data. It matters not at all whether that data is in a fielded-variable (in a file buffer), or down in string space.

CINT, CSNG, and CDBL--Convert Integer, etc.--do, however, convert numeric data. Just as the manuals say. They have been in the language since the days when we saved our programs and data on audiocassettes. They are as useless today as they were back then.

Integers are stored in two bytes. Single precision use four, and double precision use eight. To "convert" numbers from one data type to another, simply reassign the value to a variable of the type wanted. (A = B or B = I, etc.) Upward conversions always work, of course, but if the value in a source variable exceeds the capacity of the target, an overflow error occurs.

An old manual said that CINT was useful for rounding decimal fractions to whole numbers. See this:

```
 100 DEFINT C : DEFSNG B : DEFDBL A
 110 B = 1245.55
 120 C = B
 130 PRINT C
 140 C = CINT(B)
 150 PRINT C
 160 B = -1245.55
 170 PRINT C
 180 C = CINT(B)
 190 PRINT C
 200 A = 44444.55555555555#
 210 B = A
 220 PRINT B
 230 B = CSNG(A)
 240 PRINT B
 250 A = B
 260 PRINT A
 270 A = CDBL(B)
 280 PRINT A
 RUN
  1246
  1246
 -1246
 -1246
  44444.56
  44444.56
  44444.5546875
  44444.5546875
 Ok
```

Ok what? See that these "conversion functions" effectively do nothing for us. And that rounding is done without regard to the sign of the number.

Once upon a time a memory byte cost a buck. (Now they are more like ten for a penny.) Cost caused the invention of 2-byte,

4-byte, and 8-byte numbers. Presumably they persist in the big machines because of performance costs. A savings of only a few seconds of processing time each day in large DP shops can amount to as much as someone's salary for that day.

Neither of these two cost factors have much significance in today's "personal computers", but space can sometimes still be a critical issue. A large integer array requires only one fourth as much memory as double precision. Agreed. Speed is less often a design issue, however, in choosing which type of variable to use.

I use 24, 30, and 36 as yardsticks for estimating performance, for doing integer, single, or double precision arithmetic. The implied ratio is: Single precision takes 25% longer and double precision takes 50% longer than integers. It does not matter what the real times are--milliseconds, hours, days--these ratios remain about the same. The factors are approximate.

Most of the differences in time for doing arithmetic with 2, 4, or 8-byte words is caused by the time it takes to move words from memory into the MPU and back. That time does vary among micros depending on their data bus bandwidth (e.g., an 8-bit bus vs. 16-bits.) The time required for different types of calculations also varies, but that truth is responsible for a much smaller skew in the accuracy of my rule-of-thumb ratios for deliberating performance issues.

Far more critical to performance, usually, is choosing the best technique for "processing" large numbers. Those decisions do have to consider first, which data type should be used.

Assuming the need to "compute" dollars and cents--which is a valid assumption in at least 99% of my programs--begin by reviewing the magnitude (size) capacity of the different types of numeric variables:

> For integers it is $327.68- or $327.67, if insertion of the decimal point is delayed until print time, and presuming the dollars were converted to pennies in the first place.
>
> Single precision variables can handle up to one hundred thousand dollars, precisely. Unformatted, as pennies, the range is -9999999 to +9999999 (seven nines: $99,999.99). Which is enough capacity, so far, for the balance on most car loans, but not enough for doing amortization schedules

for many home mortgages.

Double precision variables can almost handle the national debt: Up to Ten trillion dollars. (Americans and Frenchmen use trillion; British and Germans say billion.) Stored as pennies, the range is -999999999999999 to +999999999999999.

Those 15 nines might be printed as $9,999,999,999,999.99--using a PRINT USING statement, dividing what is actually in the variable by 100, at print time. And that is THE KEY to how to store money in a micro, in BASIC. As pennies.

All totals should be accumulated as whole numbers. Only. No fractions. In the case of regular dollars and cents, if the input source includes two decimal positions, INT(A*100) will produce a whole number (of cents). Which makes sense.

INT (integer) works with all three numeric data types. The INT (or FIX) function should always be used when first assigning floating point numbers to single or double precision variables to preclude possible cumulative errors caused by stray bits. (Review the consequence of line 250 in the above program.)

Here is another example of what I call "stray bits":

```
DEFDBL A
A = 1234.7
PRINT A           'prints  1234.699951171875

DEFDBL A : DEFSTR X
X = "1234.7"
A = VAL(X)
PRINT A           'prints  1234.7
```

In the first case above, A is real close to being right. If PRINT USING "####.##";A is done in either of these cases, the output is the same (1234.70), and is correct, if we can ignore what is not shown.

It is embarrassing, to say the least, to have an accountant confront you with a Trial Balance listing that is in fact, out of balance by even a penny or two. Especially when it occurs after a program has been in use for several months.

The "cumulative error" in (hidden) floating point fractions can be self-compensating for a long time, over a long list of

figures being added into a single total. This is especially true where the list includes both positive and negative numbers. The risk of floating point errors is eliminated entirely if only whole numbers are used at all times.

It also simplifies the problem described in the manuals about doing IF-conditional tests involving floating point numbers. There are no minute discrepancies to worry about if no junk bits ever get turned-on.

Now is the time for some literal truth about literals; what the manuals (erroneously) refer to as "constants": A constant is a value that does not change during program execution. It may have gotten into memory in the first place from a variety of sources. A literal is a constant at the time it is literally stated in a program source line. If that statement assigns the value to a variable, whether or not that value remains constant is up to the person who wrote the program.

Imprecise English can be ignored; imprecise BASIC descriptions cannot. Take careful note: Double precision literals that do have to include fractions must also have that pesky pound-sign appendage. Observe:

```
10 DEFDBL A
20 A = -1234.7       'no data-type appendage
30 PRINT A*100
40 A = -1234.7#      'double precision, for sure
50 PRINT A*100
RUN
-123469.9951171875
-123470
```

Oddly enough, millage rates can be correctly hard-coded without having to remember to append a data-type symbol if they can fit in, and are assigned to, single precision variables. (An error of only a few mills will irk any tax assessor.)

There is another BOAB (bit-of-a-bitch) of difference between integers and floating point variables. Negative zip. Watch out for this one:

```
1110  DEFDBL A
1120  A = -A
1130  X = MKD$(A)
```

```
1140  PRINT USING "######.##";CVD(X)
RUN
     -0.00
Ok
```

Ok my foot! An Accounts Receivables statement that shows what is owed is a negative nothing can cause some customers to have second thoughts about doing business with vendors that employ incompetent programmers.

If the above program included

```
1150  FOR I = 1 TO 8
1160   PRINT ASC(MID$(X,I));
1170  NEXT
```

we would see 0 0 0 0 0 0 128 0 and can infer that BASIC does not always turn off the minus-sign bit when successive math processes result in a value of zero. George Boole would roll over in his grave if he could see this.

My compiler manual tells me that the high order bit of the third byte is the sign bit, and that, if the fourth byte of single precision variables is zero, the whole thing is zero.

It also says these things are true for double precision (viz, the seventh and eighth bytes). It does not warn me to PEEK before a PRINT USING, however. None of my interpreter manuals mention any of this, apparently preferring that we should learn for ourselves the hard way rather than back pedal (what this country boy sees as) a bug in the math package. A bug that is nearly old enough to grow a mustache.

The example used to demonstrate this not-so-funny phenomenon is a contrived one (arbitrarily assigning a minus-zero in line 1120). It does not matter. This same thing can happen when "computing" with floating point variables. It is far better to remember this simplistic example than to have a programming faux pas be seen by folks that do not trust computers.

One obtuse solution to this problem is:

```
IF ASC(MID$(X,8)) = 0 THEN LSET X = MKD$(0)
```

which cleans up all of the bytes in double precision data fields in relative files. Which can be a real problem. Many

data files have a long life span. Some records may be updated by a lot of programs, often, and remain in situ indefinitely. (This same trick works, by the way, for single precision fields by sampling byte-4, and using MKS$.)

Where care has been taken to ensure only whole numbers are ever stored, there is a simpler trick that can be used to ensure no dangling sign bits get left behind. It too is kinky looking:

```
A = CVD(X)                'A is double precision
LSET X = MKD$(A+0)        'X is an 8-byte string
```

The following ever-so-slight variation of the same trick will not work (will not clear-up a minus-zero):

```
A = CVD(X)+0
LSET X = MKD$(A)
```

It is not a bad idea, therefore, to adopt a habit of always updating floating-point fields with a plus zero, as in MKS$(B+0) or MKD$(A+0). The zero does not "hurt" the value that is being stored, and in this case, the plus sign will not run afoul of any of Judge Boole's laws.

See what is meant by remarks about the difference between what is taught in school, and what we have to resort to out here in the jungle. Adding a literal zero to anything looks as odd as that old pun about coding two STOP statements back-to-back in programs that run so fast that they sometimes skid through the first one.

As mentioned elsewhere, more than once, my preference today is to store all user-visible amounts in data files as formatted ASCII strings. Back when the capacity of a floppy was only 128kb, most of us would do almost anything to save a byte or two. Which included using MKI$, MKS$, and MKD$ sometimes.

On today's machines, with floppies holding better than a Meg, and all serious users having at least a 20mb hard disk, we can afford to spend a few of those bytes as a fair trade-off for achieving better performance.

One argument is: A simple PRINT of a fielded-variable, as in

```
PRINT X
```

is all that is needed to output what the user wants to see. Contrast this with the "old fashioned way"; something like...

```
A = CVD(X) : PRINT USING "#######.##";A/100
```

which is certainly a lot slower at output time. The extra storage cost, in this example, is two bytes--a 10-byte string vs. an 8-byte "compressed" MKD$. Two bytes times 1,000 records = 2,000 bytes. Big deal. Even, assuming five fields in each record were done this way, the cost is still only 10,000 bytes compared to 8-bytes per field the way we used to do it.

Another strong argument for burning a few of the user's bytes is that it saves him money. In programming costs. PRINT USING does a real good job of rounding, of both positive and negative amounts. BASIC can do it better and faster than we can. No matter what coding techniques are used involving ABS, INT, and FIX, or two or three IF-statements, rounding can be a real pain in the posterior for floating point values.

One caveat needs to be documented here lest you too waste hours looking for microscopic bugs: GW-BASIC and QuickBASIC do not always produce exactly the same answers when working with "long fractions".

Suppose an operator typed 4654.110217381453, and you stored it in a double precision variable called A. And later you printed what was in A using PRINT USING. The output can vary depending on the mask used, and whether the compiler or interpreter is used.

Using GW-BASIC:

```
PRINT USING "#####.###############";A prints 4654.11021738145300
```

Using QuickBASIC:

```
PRINT USING "#####.###############";A prints 4654.11021738145303
```

That little "0" (or "3") on the far right in the above two cases may or may not be significant to bankers and accountants. It may be to some land surveyors, and it is bound to be meaningful to those taking pot shots at the moon. Starwars programmers please take note.

We have witnessed a lot of growth in DOS-based BASIC. A lot of additions have been made to accommodate new mechanical gadgets. It is unfortunate that they have not opted to add, also, what would have been nice to have had all along: PRINT USING that did not print, i.e., a function for generating formatted ASCII strings, in memory, in a string variable.

PS: The best BASIC that I have ever used, undoubtedly, (and I have used better than a dozen) does have formatted-string capability. And a lot of other goodies, and, its compiler and interpreter speak the same language. It was authored by a real pro, a fellow named Ted Williams. He understands what we poor application programmers have to do to eat. From my chair, his product is analogous to the classical witticism: "The best answer is not always the correct answer." In this case, because that BASIC is not DOS-based (his O/S is called THEOS) it is not a viable option. Chapter 1 states my case for GeeWhiz. DOS is in the majority. My sense of ethics keeps me from conning my clients away from the mainstream.

Now, back to this mainstream: PRINT USING can be used to generate formatted strings, without printing. It is not as efficient as a built-in function could be, but it does work, and it is more efficient than can be done with a lot of MID$ and IF statements. The way to do it is into a dummy record, in a dummy relative file (which some dummy started calling a "random file", somewhere back down the line). The following dummy program demonstrates how this dummy does it.

```
1000 DEFSTR X:DEFDBL A                    'define data types
1010 OPEN "dummy" AS 1 LEN = 16           'relative file
1020 FIELD 1,16 AS X                      'work field
1030 A = -359550#                         'some money
1040 GET 1,1                              'reset record pointer
1050 PRINT #1,USING "######.##";A/100; 'format in buffer
```

At this point X = " -3595.50 ". Now it is easy to LSET as much of X as is wanted, into any string variable, be that a fielded variable, in a real file, or in some work area. The length of X can be set to whatever may be needed; it is a good idea that it be more than enough to tolerate the rare crap we all suffer from time to time, when a USING-mask does not define enough print positions.

Hint: This is also a simple method to use to keep the world from ever seeing "%-3595.50", and, to preclude screwed-up

columnar alignment of fields that are actually printed. By doing this dummy print first, a simple IF can be used to see if what is not yet visible ought to be, or it ought to be managed more professionally than BASIC would do it. (How nice it would be if we could error-trap this stinker. A box of TD Forms W-2 can be expensive; a line-wrap caused by this type of "bug" could mean an entire rerun would have to be done. And guess who pays for the wasted forms.)

A few more notes are needed to round off this proposed trick of the trade. The GET 1,1 is a must, every time this "function" is used, to maintain the buffer-position-pointer in the same place. So is that trailing semicolon. See why this is a dummy file: No PUT is ever done. Even so, when you close up shop, DUMMY will be seen with DIR, with no records in it. So, just before END or SYSTEM, you may want to KILL "dummy".

An alternative to the dummy trick is to get PRINT USING strings out of monitor memory. Do a PRINT USING to a known location on the screen and then use PEEK or SCREEN to get ASCII characters into a work-string. Like this:

```
X = SPACE$(20)              'DEFSTR X already done
L = CSRLIN
C = POS(0)-1
PRINT USING "#####.#####         ";A

FOR I = 1 TO 20
 MID$(X,I) = CHR$(SCREEN(L,C+I))
NEXT
```

In the above simplistic example, an arbitrary 20-byte fetch was done, and the trailing spaces in the USING-mask are for covering the eventuality that what we want has "slipped" to the right because BASIC generated a preceding percent-sign.

In the event you do not want the operator to see what is going on, PRINT USING can be done to a "blank area" on the screen with a preparatory statement like COLOR 7,7 (when the foreground and background arguments are the same, what is printed cannot be seen).

As a general rule, performance is about the same for either, SCREEN or PEEK. Using PEEK as an alternative for picking up ASCII characters from video-RAM has to contend with the every-other-byte problem, of course. (Text mode characters

are two bytes: A video attribute byte, plus the character code itself. See Chapter 6.) PEEK can still sometimes be the better alternative: In the event there is not enough "blank space" on the screen, and you have a color machine, and want to PRINT USING to a currently invisible page. (Chapter 7, ad nauseam.)

There is another trick that can be used to obtain the result of having used PRINT USING. The "numerics portion" of the binary-to-ASCII conversion that is done, is done up inside the interpreter's working storage area before the PRINT takes place. Given a default DEF SEG, PEEK can be used to fetch those characters. Like this:

```
1000 DEFSTR X
1010 X = STRING$(25,0)
1020 PRINT USING "hello ###.#####";123.456
1030 B = 1545 ; interpreter working storage address -1
1040 FOR I = 1 TO 24
1050  MID$(X,I) = CHR$(PEEK(B+I))
1060 NEXT
1070 I = INSTR(X,CHR$(0))-1
1080 PRINT LEFT$(X,I)
RUN
 123.45600
Ok
```

Several variations of this scheme are possible, of course. Here, a 25-byte work string was used, and the looping arbitrarily picks up all 24 of the bytes that the interpreter uses for constructing PRINT USING strings. When it does it, if a mask has less than the maximum of 24 digit positions, PEEK will return a zero-byte following the last-used position. (Thus the 25 zeros set up in line 1010 will ensure the INSTR trick in 1070 will always work.)

The key factor in this scheme is the address in line 1030. It really should be 1546; B in this example is shown as one less so that B can be added to I during the loop.

See the leading space at print time in this example. The first byte of the string--PEEK(1546)--could be that pesky percent sign, in the event that what is to be printed exceeds the mask.

Yes, this scheme is vulnerable (and profane). That address of 1546 is correct for GW-BASIC 3.23--other versions of GW-BASIC may use a different address. Trial and error probing can be done to confirm the correct address for a given software release. And,

needless to say, forget this scheme entirely for QuickBASIC programs.

PRINT USING is useful also for "converting" exponential notation (so called) to the format we non-scientific types prefer. And it is a way to solve those problems when STR$ will not produce our preferred format.

VAL is the "opposite" of STR$, of course, and will convert ASCII numbers back to binary numbers, regardless of their format. Even VAL is not accurately described in the manuals, however.

Numbers that have been stored as formatted strings have to dispense with commas, leading dollar signs, and the like, naturally, if you need to convert them back to numeric values. It is also necessary to be careful about what follows a number-string. It used to be that VAL would automatically terminate scanning, from left to right, when any non-numeric character was encountered. BASIC manuals quit promising that, some time ago.

Now, VAL("12 34") returns 1234, for example. Stranger still, if a string of digits is followed by CHR$(28), or CHR$(29), or CHR$(31), then VAL returns zero, no matter what the leading digits are. In GW-BASIC, that is. QuickBASIC still ignores spaces imbedded anywhere in a string of digits, but does manage to stop scanning a string when any non-numeric character other than a space is encountered, and return a true numeric value. It is best to always code so that the length-argument in expressions like VAL(LEFT$(X$,L)) is precise; if L embraces too many bytes, you may not be fond of the answer.

And we need some other answers. Answers to questions that the manuals provoke, but never explore. Viz, the permitted ranges for floating point numbers: "approximate ranges" are -1.7E+38 to -2.9E-39 for negative values and 2.9E-39 to 1.7E+38 for positive values.

Translation: These are "limits" in the sense of what can be assigned to a variable without causing an "Overflow error". These ARE NOT the ranges of what can be stored and processed correctly. (Remember, 7-digits or 15-digits are the pragmatic limits, irrespective of a decimal, or its position).

Terms can be confusing too. Mantissa refers to the bit-string that represents a number. Exponent is the (next) position in that bit-string where the decimal should be.

Floating-point numbers are stored as bit-strings, in 4 or 8 bytes, as has already been said. If we look at the 8 bytes containing MKD$(2.2) we would see (left to right in memory):

```
byte-1  byte-2  byte-3  byte-4  byte-5  byte-6  byte-7  byte-8
205     204     204     204     204     204     12      130
  in decimal

byte-1  byte-2  byte-3  byte-4  byte-5  byte-6  byte-7  byte-8
CD      CC      CC      CC      CC      CC      0C      82
  in hexadecimal

byte-1  byte-2  byte-3  byte-4  byte-5  byte-6  byte-7  byte-8
11001101110011001100110011001100110011001100110011000000110010000010
  in binary
```

The "mantissa bit string" (the magnitude) for MKD$(2.2) is:

```
10.00110011001100110011001100110011001100110011001100110011001101
```

Reading from the left, the first bit is arbitrarily turned on. This is the bit that the manuals call "implied". The next 7 bits are the 7 right-most bits from byte number 7, reading from left to right. The remaining 48 bits come from bytes 6, 5, 4, 3, 2, and 1, reading the bits in each byte from left to right.

The high-order bit in byte-7 (the left-most bit) is the sign bit for the number being represented. If this bit is 1, the number is negative; if this bit is 0, the number is positive.

Byte-8 is the "exponent byte"--it points to where the decimal would be in the mantissa bit string, if in fact it was printed as shown above. In this case, byte-8 reads 130 in decimal, but 130 less 128 is 2 (128 accounts for the high-order bit in byte-8 being on). Effectively, the exponent cannot be larger than 56, that is, the decimal cannot be beyond the length of the string of bits that represents the mantissa. (The QB manual says the mantissa is 58 bits. Best I can deduce is that it is 56 bits, the "implied" 1 + 7 + 48.)

If the exponent byte is less than 128 (the high order bit of byte-8 is off) the number is a fraction--contains no integer

portion--and the exponent number indicates how many zero-bits need to be inserted in front of the mantissa string.

Now, looking at the mantissa string shown above for MKD$(2.2), each bit position to the left of the decimal (the integer) represents a positive power of two; each bit position to the right of the decimal (the fraction) represents a negative power of two. As in 1, 2, 4, 8, etc. If a bit is one (1) it is on; if it is zero (0) it is off. Notice that the integer mantissa bits are stored from right to left and the "leading zeros" are dropped (the number is "normalized" according to the manuals). The fraction mantissa bits begin immediately after the "binary point", and are stored from left to right.

The following shorty demonstrates how to "read the bits". In this case M contains a string of ASCII ones and zeros that correspond to the mantissa string for MKD$(2.2).

```
1000 DEFDBL A:DEFSTR M:DEFINT C-L
1010 M="1000110011001100110011001100110011001100110011001100110011001101"
1020 E = 2 'Exponent (note decimal position is "implied" by E)
```

To compute the integer-portion of the mantissa:

```
1030 G = 0                                              'G = powers
1040 FOR I = E TO 1 STEP-1 : C = ASC(MID$(M,I))-48      '0 or 1
1050  A = A + C * 2 ^ G : G = G + 1
1060 NEXT
```

To compute the fractional portion of the mantissa:

```
1070 G = -1
1080 FOR I = E+1 TO 56 : C = ASC(MID$(M,I))-48
1090  A = A + C * 2 ^ G : G = G-1
1100 NEXT
1110 PRINT A
RUN
 2.2
Ok
```

Although an exponent of -127 is mechanically possible, the "limit" of 2.9E-39 cannot be seen, per se. The books say PRINT USING can specify a "mask" of 24 digits; in fact that limit is 23, for fractions, because the decimal point counts as a "digit" in a USING mask.

So: -2.9D-17 will PRINT as 2.775557561562891D-17 or, as .00000000000000002775558 with a maximum allowable PRINT USING mask of ".######################" (23 pound signs). The other "limit" of -1.7D+38 will print as .4995850990994722, or it will be the same string of digits followed by 7 zeros if a max-size PRINT USING mask is used.

All of the above is true for single precision, which uses only 4 bytes for storage, but 8 for internal machinations (then the number is "truncated").

Here is another way to observe the internal similarities of single and double precision numbers:

```
1000 DEFDBL A:DEFSNG B:DEFSTR M-Z  'define data types
1010 M = MKD$(1234.7)              'LEN(M) = 8
1020 X = MKS$(1234.7)              'LEN(X) = 4
1030 PRINT (X = MID$(M,5,4))       '-1 if true, 0 if false
1040 PRINT CVS(MID$(M,5))          'same as X
RUN
-1
 1234.7
Ok
```

See that the four right-most bytes of double precision numbers (8 bytes) are formatted the same as for 4-byte single precision numbers. Sometimes we can take advantage of this "fact". Like this:

Assign a number to an 8-byte string, as in

```
M = MKD$(1234.7)
```

then test to see if it will fit in only four bytes, with

```
IF LEFT$(M,4) = MKS$(0) THEN....
```

which will allow us to "pack" numbers in fewer bytes in data files. Sometimes. Of course, when reading such files some mechanism is needed for determining which "fields" are 8-byte numbers and which are four.

Meanwhile, back to the story of the "phantom bits". See why IF A=0 may not work, even when we PRINT A and see it is zero. When working with floating point variables it is imperative to

remember that many bits may be hidden from the casual observer. And hidden from casual (or amateur) programmers. And from some teachers, preachers and politicians, casual or otherwise.

There is also a "bit" of difference between those two BASIC language products (GW-BASIC and QuickBASIC) that are supposed to produce identical results. When fiddling with minute fractions be aware that MKD$ and CVD may not always translate as exactly the same number. See this:

```
1000 DEFSTR M:DEFDBL A:DEFINT C-L  'define data types
1010 A = 6638.071237449458#        'double precision number
1020 M = MKD$(A)                   'LEN(M) = 8
1030 FOR I = 0 TO 15               'alter left-most byte
1040  MID$(M,1) = CHR$(96+I)       '96 = 60 in hex
1050  PRINT HEX$(ASC(M)),CVD(M)    '60, 61, etc.
1060 NEXT                          'until 6F in hex
```

RUN with GW-BASIC produces

```
60          6638.071237449458
61          6638.071237449458
62          6638.071237449458
63          6638.071237449458
64          6638.071237449458
65          6638.071237449459
66          6638.071237449459
67          6638.071237449459
68          6638.071237449459
69          6638.071237449459
6A          6638.071237449459
6B          6638.071237449459
6C          6638.071237449459
6D          6638.071237449459
6E          6638.071237449459
6F          6638.07123744946
```

but if compiled with QuickBASIC this program produces

```
60          6638.071237449458
61          6638.071237449458
62          6638.071237449458
63          6638.071237449458
64          6638.071237449458
65          6638.071237449458
66          6638.071237449459
67          6638.071237449459
```

```
  68          6638.071237449459
  69          6638.071237449459
  6A          6638.071237449459
  6B          6638.071237449459
  6C          6638.071237449459
  6D          6638.071237449459
  6E          6638.071237449459
  6F          6638.07123744946
```

which at a glance seems to be the same output. Look again.

Given that M is an 8-byte string, and that the left-most byte contains the least significant bits of MKD$(A)--the trailing decimal digits--then look what happens when that byte is equal to 65 (hexadecimal). GW-BASIC "reads" this number to have a final digit of 9. QuickBASIC says it ends in 8.

Now, suppose you saved some numbers in a file with GW-BASIC. And you read those numbers back with a compiled program. Or vice versa. What you see may be different than what you saw.

The moral of this lesson is the seesaw. Be alert when "mixing" these two languages. It is nice to be able to do dynamic debugging in interpretive mode of programs that are to be used later in compiled form, but watch for gotchas like this. And remember that 2 plus 2 is not always 4, exactly, in BASIC (or a lot of other languages).

Reviewing what I have just written, I am compelled to request your indulgence, but not inclined to rephrase anything. No offense is intended toward those who teach. Their world has real-life demands too. For sure.

The systems software products we are obliged to use could always be better too. For sure. Their developers are often forced to abide by economics-based decisions, no doubt, even when their expert acumen would dictate otherwise. Granted.

What has been shown is how this old duffer has learned to live with what exists. Critical cracks are meant to give perspective; coding suggestions are meant to be useful advice. Sporadic levity is meant to keep you from falling asleep.

None of this should be construed as recommendations to use anything other than "good programming practices". In my world, what is good, is, usually, what is profitable. If you too can

profit from any of this, neither of us has wasted our time.

Chapter 6 = DEVICES

When we have to write a program that can run in more than one environment--on various makes or models of machines, varying machine configurations, or updated software versions--then we really begin to appreciate the challenges of interfacing with the outside world. Challenges for us, and for all language developers.

Most language developers try to remain aloof and avoid direct contact with hardware devices, preferring to do all input and output via "device drivers" in the operating system. Being one rung further up the software social ladder, programmers like us prefer not to consort with the peons at the ends of the cables, either. But, we have no choice. We live in the real world. Systems software writers seemingly live in urban cloisters.

Granted, our cultural positions are better today, in general, than that of a few years ago. Nearly all "dumb devices" are now "intelligent peripherals". But, they still enjoy being unique. Sometimes their uniqueness goes beyond a class-level; some individuals born only a few days apart will have traits that resist legislation intended for the good of the majority.

Add to that observation that it often seems there is no single legislature. In fact, one is tempted to think sometimes that there is social conflict between computer manufacturers, the language developers, and the operating systems writers. They each tend to write their manuals autonomously, only now and then alluding to the vagaries of the others.

Which brings us to the point of this chapter. We can presume those folks were paid for what they did. Our reward depends on making an application work. Not many clients will be very sympathetic to an excuse that "the manuals" are vague, remiss, or even wrong. The buck stops here.

Witness: Wrote a nice little application for the bookkeeping department of a company that owned several convenience stores. They called it an inventory program. It was so simplistic that even that name seemed highfalutin. But it did work, and did what they wanted it to do, for a long time.

Then one day, they hired a new operator. One that "knew a lot about computers". She wanted to fancy things up a bit. She held down <Alt> and punched the number for a pseudo-graphics

character. Know what happened when she let go? Crash-o. The next output on the monitor was the operating system prompt.

There she sat with files left open, buffers loaded, data files half updated, the FAT fractured, and no backup since last week. My program was gone. The interpreter was gone. The operating system was in limbo. The operator was mad. And my immediate thoughts were about voodoo dolls and the natives that write systems manuals.

The BIOS in that machine crapped-out on any <Alt>+number. The keyboard was an older 83-key model. I finally deduced that the BIOS would only tolerate the newer 101 models. Nothing in any manual said so, yet, the machine was still configured just as my client got it from a computer store. (Rather than spend a lot of time developing a BIOS patch, I opted instead to simply give them the 101 keyboard that I had used while developing their program. My out-of-pocket cost to fix "my bug" was sixty bucks. Lab time-to-find was over ten hours.)

Now, back to once upon a time, that time when the interpreter itself was "the operating system". When you cranked up the machine, it came up running BASIC. It was in charge. It could predict what would happen when you used its commands. Today, it no longer has autocratic authority.

Today the interpreter is just another program. It is now a software peer of word processors, spread sheets, and games. Any of these may run in the "domain" defined by the operating system; all of them are supposed to live by the laws of that kingdom. Most are law abiding. Most of the time.

When it comes to files and devices, when it is necessary for the interpreter to make calls to operating system services, it can promise you what the outcome will be, only to the extent that it can assume it will regain run-time control. And you can write your operator's instructions from a third world only to the extent that when (if) your program regains control you know, exactly, what is what, where what is, and what might or might not have happened behind your back.

Remember that operative word IF when you read the following suggestions. Students trying to get an A in Computer Sci 101 do not have to worry about Murphy's law. If you are getting paid to write a program to do payrolls, much of the code you write, most of the documentation, probably over half of the

testing, and a predominant influence on overall design has to deal with precisely that. What IF.

If you want a punch-proof keyboard, good luck. We would like to design "turn key" applications secure enough to promise an operator that they can do themselves no harm, no matter what keys they hit on the keyboard. So....

If you have a nicely laid out data entry mask on the screen, and the operator is supposed to fill in the blanks:

INPUT will not work, obviously. If the wrong key is hit the interpreter will clobber your nice layout and tell the operator to "Redo from start". Which makes no sense. The last thing you want them to do is turn off the machine and turn it back on. Meanwhile, the cursor is no longer sitting where you last put it.

LINE INPUT will not work, either. If a cursor-arrow key is hit, all kinds of things must be done to clean up the mess on the screen. Not to mention the problems of, if they type too many characters, etc.

INP can be used, but it is a bad idea. It is in the same league as PEEK and POKE; bit-fiddle in toy programs, but not in real applications that may have to survive changes in machine configurations, new models, or software updates.

INPUT$ might be used, one character at a time, but not if you want to see what is in the second byte of two-byte key codes.

INKEY$ is the remaining choice. Use it inside a WHILE/WEND loop, echo what you want, accept as many or as few keys as are wanted, and BEEP when you want to. Yes, this solution requires several lines of programming, at a minimum. Yes, you are effectively having to write your own keyboard driver. With it set up as a subroutine--or a couple of specialized ones--the overhead is not too bad. If efficiently written, performance will likely be acceptable. Chapter 13 contains suggested techniques for implementing this concept.

If you want a punch-proof keyboard, good luck. We would like to design "turn key" applications secure enough to promise an operator that they can do themselves no harm, no matter what

keys they hit on the keyboard. But....

If an operator hits <Print Screen> and the printer is on, and on-line, that request is going to be carried out without your permission, or even, your awareness. And you thought the printer was on line 40 of a 66-line page, maybe. Or you had been counting down to the top of the next W-2 form, or whatever. Meanwhile, the operator just dumped your menu all over their pay checks. And destroyed the continuity of your next-check-number logic.

If an operator holds down <Ctrl> and hits <Break> you may or may not know it. The interpreter may go immediately into program editing mode, or, it may trigger an ERROR 8. Which can confound your error handler, because you know there is no "Undefined line" in your program. And you cannot simply BEEP and RESUME NEXT. (See Chapter 9 about this, and other strange encounters in the never-never land of error traps.)

If an operator holds down <Alt> and <Ctrl> and hits <Del>--horror of horrors--we all know what happens. Just try explaining to some, however, why they can do this on a whim in a card game, but they had better not do it while posting accounts receivables transactions.

We can remap the keyboard, of course: Reassign the meaning of some keys, and turn-off some, altogether. But this is a bad idea. It is too specific. It requires knowing exactly, what keyboard, which operating system version, and what BIOS-clone is being used. Which makes it tough to transport any program to another machine, or survive upgrades.

On any (regular) request for keyboard input, the interpreter translates whatever key-codes the underlying hardware and software hands up. If we stay within the realm of BASIC, accepting its definition of keys, it is reasonable to presume that future upgrades of the language will not obviate what worked in the past.

PS: Some manuals encourage you to use ON KEY trapping; some include some "scan codes". In my shop we don't use either. This trap is as mean as a bear trap, and, the codes shown are seldom the right ones for a given machine.

Similarly, if we use INP, the interpreter simply grabs a byte from the requested port and hands it to us. It cannot

do any translation for us. (The purpose of INP is to get an unadulterated byte from a device port without any regards as to what type of device is connected to that port.) These are also the codes we get if we opt to OPEN "KYBD:"--which still makes our program sensitive to specific keyboards, the BIOS, and different releases of device drivers.

My money is bet on the interpretation of keys by BASIC. Sure, it chooses to ignore some, like <F11> and <F12> and the seldom used <Scroll Lock>. And it gives me the same codes for those duplicated gray keys and white keys. If the operator wants to move up one line on the monitor, it matters not to me which up arrow she hits. K.I.S.S.

"Keep It Simple, Stupid." That advice is echoed in nearly all books about programming. Suffer my passing it along one more time. It is fundamental to my suggested solutions to keyboard input problems.

To minimize the risk of an unwanted <Break> tell your user that they should never use <Ctrl> or <Alt> while running your program. According to KISS, there are too many keys on the keyboard as it is.

In any long running process, like a sort routine, output some indication from time to time that your program is thinking, and not on cloud nine because the <Pause> button was hit inadvertently.

A warning about the use of <Print Screen> should be enough. Untimely use of that key is a risk, but, no more so than the many other things that can go wrong while trying to control a printer. Which is a separate topic, later.

And print a character. Or move the cursor. Or change the page. Or blow the horn, or something, so that they soon get used to the idea that they will get some response from all live keys, and the dead ones do nothing.

That last suggestion sets the tone for a few notes about that real odd ball device: The monitor. In my book it deserves only a few notes. To cover all of its variants would require a hefty tome, indeed. At any one point in time, probably only two or three pages of such a book would be relevant to a given application problem.

If we tally the current list of key words in BASIC we find that over ten percent of them are unique to this one device. Added to that, old-timers like LOCATE have been stretched. Once upon a time it required only two parameters: line and position; or row and column, if you prefer. Now as part of LOCATE, you can turn the blinking cursor on and off, and you can specify what the blinking block's size should be. And you can omit any of the parameters that need not be changed. So the manual says.

Exterminators beware: There are bugs in that advice about not having to restate line and column parameters. Try LOCATE ,,1 to turn-on the cursor, after having last printed something on line 25 at position 80. ERR = 5 is apt to occur. (Chapter 9 is the one, remember, about many types of strange encounters.)

Not to malign the eloquence of Sir Winston Churchill, but: "Never has so much been done by so many, to accomplish so little." Nay, this is not the Battle of Britain, but neither need it be a battle of the bytes.

Fancy output takes a lot of code, a lot of time, and delimits the environments in which a program can run. Granted, programs that are to be mass marketed have to have sex appeal, same as boxes of cereal on grocery shelves. If your client thinks he wants his masks and menus in blue, and green, and yellow, and red, he may change his mind when you tell him that each change in color will cost a couple of hundred dollars extra. When you tell him also, that high intensity doesn't work on some brands of monitors, and that underlining will cause blurred text on some, he may change his mind about the worth of pretty.

Here are my stock solutions for "conventional" monitor output. They appeal to some, appease most, and keep my labor charges reasonable. They are sufficient for all data processing programs. (Pixel pounding is too far afield from the subject of generic devices. Chapter 7 is dedicated to the graphics gang.)

Write all programs so as to be self-adapting, to whatever adapter is in the machine. My favorite technique during program initialization:

```
  CLS                                   'so that it is blank
  COLOR 0,0                             'invisible
  PRINT "oops"                          'at position 1,1
```

```
BM = &HB000                              'Base Monitor address
CM = 12                                  'Cursor Max size
DEF SEG = BM                             '1st assumption (mono)
I = PEEK(0)+PEEK(2)+PEEK(4)+PEEK(6)  'hash = ASC(o+o+p+s)
DEF SEG:COLOR 7                          'reset the defaults
IF I-449 THEN BM = &HB800:CM = 7     'true = color adapter
```

Note: An alternative trick is to do something "illegal", like PCOPY, for an assumed mono-adapter, then let your error handler set a flag, then do a RESUME NEXT. This can be a little tricky, however. WIDTH 40 will trap in interpreted BASIC (on a mono-adapter), but not if the program is later compiled. Read Chapter 7 for more about differences in SCREEN modes, and about how the compiler and interpreter differ when addressing different monitor adapters.

From here on, BM is the DEF SEG value for the monitor's memory, and CM is the second of the two values for sizing the cursor, zero being the first value, of course.

When the interpreter is first loaded it self-initializes in text mode--WIDTH 80:COLOR 7,0,0--with an underscore-like cursor. Because data entry operators glance up only now and then, a larger (full block) cursor--LOCATE ,,1,0,CM--is easier to spot quickly. During field editing, when their focus is totally on the screen, LOCATE ,,1,CM\2,CM is good (a half block) for indicating that they are in insert mode, for example.

A note is needed about a seeming contradiction: DEF SEG names a "hard address"; any time we resort to that we are vulnerable. Some compromises are inevitable, however. This one is based on calculated risks. Having stored that address in a variable, in only one (fixed) location in all programs, even if it does have to be changed some day, that effort should be minimal.

PRINT lesson: The character fonts are not the same. This can be seen in the hardware manuals, but not until it has been seen on the screen, do you see what they mean.

Because the characters displayed by a mono-adapter are formed in grids that are 9-dots per print-column, and 14-dots high, per print-line, they are much crisper than a Color Graphics Adapter's 8 x 8 format.

Because of this, reversed video has to be placed low on your list of options for highlighting text. In mono it can be done nearly anywhere, any time. To achieve an acceptable degree of legibility with a CGA, the print-space above highlighted text must also be in reverse video. Even if that space is just that, i.e., space-characters. (Thus, do not do reverse video on line one.)

Another useful option in mono is underlining. Forget it, if you want to design screens that will work on anybody's machine.

Agreed, the above is basic to hardware, not to BASIC, but it can still be an expensive lesson to learn the hard way. Like the one about the border-argument for COLOR. Never use it, either. Especially in programs that need to survive the next upgrade (to EGA/VGA, for example).

There are three ways to talk to the tube. Conventionally, using LOCATE, PRINT, and the like; POKE bytes directly into monitor memory; or, OPEN "CONS:" for OUTPUT as a file.

The third possibility is now nearly archaic. It can still be done, but to no great advantage. In old machines (and still on big mainframes) it was a way to output to video terminals. Asynchronous devices. Where, when you PRINT, the output is sent down a cable. Where, the device at the end of the wire, printer, boob-tube, or otherwise, outputs characters itself, or does mechanical functions such as a page slew depending on the codes that are sent to it.

A monitor is different than a terminal, although, this is hard to explain in ten words or less. The picture on a CRT (Cathode Ray Tube) display is being constantly repainted by electron beams scanning over the face of the screen. A page on the screen correlates to a block of memory. When we do output to a monitor, we put codes directly into bytes of the computer's memory. On a CRT-terminal, the codes are stored remotely, in the terminal's own memory.

Although an operator thinks they look the same, programming a monitor can be done quite differently than for a terminal. Knowing we have a monitor, and knowing which page of memory is being displayed, and knowing which memory bytes align with which row and column on the screen, we can POKE codes just where we want them. This results in what will seem to be, an instant change on the screen. And quick we must be.

Contemporary packaged programs such as word processors are hard to mimic when it comes to doing data processing jobs in BASIC. But, they have set a de facto precedent for operator interaction with the machine. Granted, some are far superior to others. Some adhere to basic principles of human factors engineering; some were written by programmers unfamiliar with ergonomics, biomechanics, and typography. Collectively, they have become what is expected: "All programs work this way."

So mine work this way: All screens are BLOAD files. They are built separate from the using program. Rather than having to do LOCATE and PRINT statements a zillion times, which is slow, and tedious, BLOAD is used like a shotgun. Not only is it fast and easy, a lot of text can be blasted with a single program statement. Working in text mode, here are some fundamentals: (Again, see Chapter 7 if you like to punch graphics pixels.)

Each character on the screen uses two bytes of memory. The first byte is a character's code, itself. The next byte has the preceding character's COLOR attributes. A screen shows 25 lines of 80 characters. A line in memory is 160 bytes; a full screen in memory is 25 times 160, or 4000 bytes.

To quickly blast a screen "mask" do a DEF SEG = BM followed by BLOAD "filename",0 (assuming BM has been pre-loaded with either &HB000 or &HB800 as shown in the earlier example).

The offset parameter following the BLOAD is critical. If a BSAVE screen was built on a mono-machine and the target one has a graphics adapter, and you omit the offset when you do a BLOAD, the file is aimed at the wrong place in memory. It is loaded at the address specified in the file-header, i.e., the address from which the BSAVE was done. (All BSAVE files have an extra 8 or 15 bytes added to what you save. More on that in a minute.)

By the way, if you do a BLOAD into a nonexistent block of memory, no one will tell you. The bytes just zoom into the ether.

In the program that builds a screen, a mask, a menu, or a help page, do DEF SEG = BM followed by BSAVE "filename",F,L with F and L specifying From and Length. To save an entire screen F = 0 and L = 4000. For partial screens it is easiest to work with whole lines. To BSAVE lines 5 through 10, for example, to jump over the first four lines, F = 4*160 and the

length (to save) is, L = 10*160-F.

While a program is running--while an operator is entering data in fields on a mask, for example--an existing page can be saved, as is, before doing "pull down" overlays such as help screens. Do a BSAVE of the as-is screen, BLOAD the overlay, and when ready to resume, BLOAD the one that was saved in the hold-this-picture file. In this case, because the program that is doing the BLOAD is the same as the one that did the BSAVE, no offset needs to be specified with the BLOAD.

Another by the way: BSAVE can be a little slow, depending on where you send the file. Going to a floppy is slow; hard disk is much quicker, but the very best place to hold work screens is VDISK. That is quick, both for the BSAVE, and the BLOAD to bring it back. Forget PMAP, PCOPY, VIEW and their kin, if you want to run on any machine. (Sales of monochrome monitors is likely to continue to outpace color. They still cost a lot less.)

Most of the above can be deduced by carefully reading all of your systems manuals from cover to cover, in maybe something less than thirteen passes. Here are some tidbits garnered by trial and error; some errors can be real trials. Given this hindsight, on your next pass through the manuals watch for these. They are mostly there, even if not easily discerned.

The first byte of a file that was created by a BSAVE is a file-type indicator. It is equal to 253 in decimal. It is followed by three, 2-byte words.

The first pair of bytes following the 253-code are equal to BM, the value you used with DEF SEG = BM. The next pair of bytes are the offset, and the next pair are the length that was specified with BSAVE. (In machine language context, the bytes in 2-byte words are reversed, remember.)

The 7-byte file header is followed by a continuous string of bytes, just as copied directly from memory to disk. Which means, for text mode files, a character-byte, followed by its color attributes, followed by the next character, and so on. Ergo, what was on the screen, colors and all, blinking and what have you, can be brought back to the screen, just as it was, at the moment the snapshot was taken.

Incidentally, the memory-image file block used to be followed by a copy of the same seven bytes that are in the file's header, plus one more byte, a control-Z, code-26. In some recent release, three-point-something-or-other, that extra 7 bytes at the end no longer occur. Find that little gem in a manual, if you can. (Another example of: If they never told us in the first place, they can skip telling us they changed it.)

Two more notes need to be made; they are in the books, but, not noticeably so. COLOR ,,(border) is a global aspect. It relates to an entire screen; it is not an attribute of any one character so it is not saved in a BSAVE file. (PS: Do not use the border-argument. Forget it. You can spend many hours trying to fix programs that are moved to a machine that has a different monitor adapter.)

The other note at hand is, think of the cursor as a figment of a monitor's imagination. The cursor's location, size, and current on or off status are all maintained by adapter-driver software; it is not a character, per se, in a byte, in memory.

Hint: I have been known to save these values with my BSAVE files, anyway. Reserve a few blank spaces somewhere. Make them hidden as in COLOR 7,7 then PRINT CHR$(value-to-save). After a BLOAD, use SCREEN or PEEK to retrieve those values. (See Chapter 12 also, about using the monitor's memory for storing data, and for inter-program communications.)

Given the above, the possibilities are numerous. It is not a very big chore at all to write a program that can make BLOAD files out of text files. Or vice versa. Or to write filters that can change the colors of existing masks and menus, a real solution for transporting some applications from CGA monitors to monochrome, for example. (Like for converting what was blue, to bright, maybe, to get rid of unwanted underlining.)

Chapter 12 tells my methods for easily creating BSAVE masks in the first place. Getting back to the theme of this chapter, here are a few more notes about making a monitor act like you programmed in C, or some language that claims to be "closer to the metal" than interpreted BASIC.

Moving highlighted words: A lot of menu techniques let the operator select a function by using the cursor-arrow keys to change which text string is currently being displayed in

reversed video (or underlined, or a different color, etc.)

Conventional BASIC would expect you to do a COLOR, a LOCATE, and a PRINT (to do not-highlighted-video) of the text you are moving away from. And then do, another COLOR, a LOCATE, and a PRINT of the moved-to text. That is slow. No matter what souped-up machine you are programming on.

Mine is a much faster trick, unconventional though it is: Calculate the offset (DEF SEG = BM) to the address of the first highlighted character's attribute byte. Use FOR/NEXT with a STEP of 2 and POKE the code necessary to turn off the highlight. Now calculate the offset to the address of the target area and use a 2-step FOR/NEXT again, to POKE the attribute bytes of the new string of text to be highlighted.

See that POKE to a character position is similar to PRINT. And when you POKE to attribute positions you are doing what the interpreter would do with the first two values used in a COLOR statement (but it doesn't do it until you PRINT).

Similarly, a SCREEN (function) can be mimicked with PEEK, but faster. When you specify two parameters with SCREEN the code that is returned is the same integer value that you would get using PEEK addressed to a character-byte. A third, non-zero parameter in a SCREEN function will return an integer value corresponding to the bit pattern of an attribute byte, the same as would a PEEK to that same address.

Granted, if you are accustomed to using LOCATE to get to a line and a position, and have developed instant recall for the digits to use to get the COLOR you want, PEEK and POKE tricks may seem more complicated. Initially, anyway.

Lines and positions are not difficult to calculate if you use your imagination instead of looking at the screen. In your mind's eye see that each line is 160 bytes, not 80. The codes for characters are in even-numbered positions (beginning with position zero). The color of each character is in the next adjacent, higher numbered byte.

If you enjoy bit banging in BASIC go ahead and unscramble the attribute bytes so that you can see what numbers would be used if they were created by a COLOR statement. But you don't have to. Just PEEK a byte that already has the desired flavor and make a note of that number; it is the seasoning to use in a

POKE statement later, with no impact on your gastrointestinal system.

Meanwhile, lose no sleep worrying about being a hypocrite. My constant inference that PEEK and POKE are vulgar must be viewed in perspective. Used within the monitor's memory, they are tolerable, even if not Platonic. They still should not be used in mixed company. Bit fiddling elsewhere, in programs built for others, is still considered sinful on my farm.

> Reminder: DEF SEG = BM is critical before doing POKE or BLOAD, or you are apt to get shot right out of the saddle.

Putting data on the screen, quickly, is a horse of a another color. Whether we are working with COLOR or not. PRINT is the pragmatic choice, but, the fewer the better. Consider:

```
PRINT X$;:PRINT Y$;:PRINT Z$
```

This could be done as PRINT X$;Y$;Z$. Obviously. What is not obvious is the performance difference for singular instances. For a one-liner, on one screen, followed by interaction with the keyboard clerk, the performance differential of these two alternatives is insignificant. For twenty lines in succession however, the difference is very noticeable.

For example: X$, Y$, and Z$ are fields in records; they are for the account code, name, and phone number of customers. On screens that halt for operator input after each one is shown, no problem. If you have to display say, twenty records per page, your operator will appreciate faster page-paint time. Count the difference. Sixty PRINT statements vs. twenty.

Better still, use one variable instead of three. Chapter 3 tells how long it takes the interpreter to find your variables; common sense is enough to know that the time for twenty has to be less than for sixty. This is a good example of where common sense can produce better results than textbook doggerel.

If, in the above case, X$ and Y$ and Z$ are fielded variables in records, named one after another in a FIELD statement, the record could be redefined so that one variable encompasses all three fields as a continuous string.

Before pondering the question of what to do if the fields that are to be displayed are not juxtaposed in the records, ponder

first why they are not. Who designed the record layout? How was it decided which fields should come first, second, and so on? When it comes to adding to an accounting balance it does not matter where in the record that data field is; not many people watch a computer compute. When you are outputting data to a monitor, however, somebody is going to be watching.

Now, extend the above two suggestions all the way: Arrange data in records in the order they will most often be shown. Store numeric amounts as strings, already formatted. To heck with PRINT USING; do the "using" when you do the computing, and store the result in the records so that it can be shown as is. For blank space between fields, put those spaces in the records also. With the capacities of today's disks we no longer have to be as stingy as we once were. (Chapter 5 tells how to save formatted numbers in data records, easily and quickly.)

My favorite payroll application has a master-file layout with 1200 bytes in each employee record, mapped as 103 data fields. Only one PRINT statement is needed to display an entire record. It blasts three lines for each of five (240-byte) variables. That is quick. It is nearly as fast as a word processor that is holding all it knows, in memory.

> PS: Multiline PRINT statements can be tricky. Printing begins at the column where the cursor is currently sitting. Supposedly. Or, it commences in column one of the next line down if what is to be printed is "...wider than the screen". Predicting just what "wider" means is not easy. Compiled programs behave differently than interpreted ones: BASIC, BASICA, GW-BASIC, and QuickBASIC all play by different rules when you are near the bottom right corner of the screen.

Back to payroll: A BLOAD "mask" is done first, with all the pretty lines, and boxes, and field descriptors. The cursor is sent to line 1, column 1. GET loads my five fielded variables, then I do a one-shot PRINT U$;V$;X$;Y$;Z$. Which outputs a total of fifteen 80-character lines of data.

Sure, there is some waste. To make adjacent fields straddle my pretty lines, they are separated by a CHR$(28) byte, which works as a right-arrow key would. To skip over column-header lines on the mask, line-advance codes are permanently imbedded in the records at fixed locations. Total waste is about 200 bytes per record. For 100 records, it is about 20,000 bytes. Which is less than one percent of a 20Mb hard disk. If that

still seems extravagant, do some counting of the junk bytes in a typical word processing or spreadsheet file.

This concept of ready-to-show record formats has to consider that other "standard output device": The printer. You say LPRINT or PRINT #<file-number> and a string of bytes--codes and characters--get sent down the wires, one at a time. The interpreter sends whatever you tell it to, and, it sends some codes of its own choosing. Like, after each print statement (that has no trailing semi-colon or comma) it sends a pair of bytes, to tell the printer to jack the paper up one line.

The <Print Screen> function works similar, but the data that is output comes from the monitor's memory. It is sent by the operating system, not by BASIC. They do not work in exactly the same way. Nor do they know what the other is doing, how, or when.

The interpreter is (marginally) smarter than the operating system. You can use WIDTH (printer) in a program, or OPEN the printer as a "random file", to pre-condition the interpreter to know when to, or not to, issue line advance codes.

The <Print Screen> key invokes a direct service call to the operating system. It has no idea what is going on inside the interpreter. It outputs 25 full-length, 80-byte lines, all 2000 of the character-bytes in a monitor memory page, skipping over the attribute bytes. Which explains why, by the way, hidden characters on the screen will not be hidden when they get to the printer.

What happens at that end of the line is another matter still. Even the cheapest printers are today, "intelligent". So much so, you have to be clever to get them to do what you want, and you have to be careful or they will outsmart you.

Because there are so many makes and models available, and they seem to get smarter every year, trying to anticipate what will happen is not unlike trying to predict what may happen when a teacher has to leave the classroom momentarily. How to best cope with adolescent children is a subject for experts. The suggestions that follow are merely for unruly printers.

Always OPEN the printer as a file device and do output to it via PRINT #<file-number> statements. The alternative is to spend several days trying to figure out why LPRINT is causing

odd "File already open" errors. This must be triggered by, apparently, an internal conflict related to doing BSAVE/BLOAD operations. Whether this is the true culprit, or common to all versions of the software is unknown to me. At a point I quit experimenting with kluges to get around the problem and just accepted that LPRINT was practically useless.

Cardinal design rule: Plan all print lines to be one byte less than the width of a full line. On an 80-character printer, for example, format your lines to print only 79 columns. This is a pragmatic rule. Too many others want to help decide when it is time to do a line advance. The interpreter has its own screwy and inconsistent ideas. Most modern printers try to get in on the act also, but each has its own theory about what is best.

Far less time (and paper) will be wasted during testing if you keep it simple and use a semicolon to continue-on-a-line, or no semicolon to print-and-advance, never letting the printer get to column 80 (e.g.) where someone else is apt to contradict your intentions.

Printer programming maxim: Portability problems precludes all pretty printing. The "Install" programs supplied with many packages are sometimes bigger than the end-use product because there are so many different printers out there (that must be programmed in so many different ways). Aesthetic output is a demand of word processors. For my money, accounting reports can be fully useful rendered in simple ASCII. Pretty printing can cost a pretty penny, and have more impact on the bottom line than can be cost justified.

One bent coin, a CHR$(27), can cause more problems than a sack of slugs in a casino. Escape code sequences date back to the earliest days of mechanized data processing. The tradition continues today that a byte that looks like 1B in hexadecimal is to be seen by a receiving device as a warning that, what comes next are device parameters, not printable characters. See this:

> On a monitor, a CHR$(27) is an attractive left-arrow symbol. Sent to a printer, with <Print Screen> or otherwise, it can reek havoc. Transmitted unintentionally, the next one, or two, or several bytes, are bound to be misunderstood. Even printers that are supposed to be "compatible" may switch to italics, underlining, or whatever, differently. All of them

will react to an escape-code sequence--not many will react in exactly the same way, for a given series of codes.

This is no less true whether coming from the monitor, or sent inadvertently in a string of data coming from files or coding bugs. Caveat. In Latin, English, or any language, a code-27 byte is not profane to a machine, but it can provoke strings of profanity from humans.

The other low-order codes in the ASCII chart--those below a space character, CHR$(32)--are less pernicious. Some may cause only minor problems; some, severe migraines. Sticking to decimal notation, here is my general attitude toward these naughty kids.

Code-12 is one of the most dependable. It is useful to slew the paper to the top of the next form or eject a page. It must be used consistently, however, to achieve consistent alignment on successive pages. If CHR$(12) is followed by a semicolon, the next thing printed begins on the current (now top) line. No semicolon (or comma) after CHR$(12) produces one blank line after the slew, because, BASIC sends a carriage return code after your top-of-form code.

Code-28, as cited in the example earlier, has not caused me any grief thus far. On the monitor it bumps the cursor one position to the right, like a nondestructive space character. To date my experience with a variety of printers has been that this code is synonymous to a code-32 space character.

Code-13 is usually followed by code-10; seen in hex as 0D0A. This pair is what the interpreter sends for you, to cause a line advance and a repositioning of the print head to the beginning of the next line. If these codes are imbedded in your records, you may need to use a semicolon after the last variable, to keep from getting an extra line advance.

PS: That old green manual--circa 1986--is misleading. It says BASIC sends out only a CHR$(13) and leaves it up to the device driver to send the follow-on CHR$(10). Makes no difference to me who is doing it. Fact is, when using "standard" software, all sequential output--to a printer or a data file--always includes the 13/10 pair of codes at the end of each line.

Avoid all of the other codes below 32 unless your boss is liberal with overtime. Code-0 is a funny one, for example.

A zero-byte will be seen on your monitor as a space, same as a code-32. Printers have minds of their own, remember. Some will also produce a space. Some just discard a zero-byte, pretending it was not received at all.

The high-order codes--those above 127--can be brats also. Especially those from 128 through about 155. Notice that 128 plus 12 is 140. Or conversely, in terms of bits in a byte, if the left-most bit is ignored, decimal 140 becomes 12. Thus, what looks like a lower case letter (i) wearing a hat, on the monitor, causes a page slew on a lot of printers.

In short, if you want to always know what column, what line, and what page you are on, no matter what printer is plugged in, be cautious of the codes you send. Those from 32 through 127 are pretty predictable. The others may be pretty, but pretty onerous when used in peculiar circumstances.

Thus ends my notes on what can or cannot be printed with confidence. Which assumes that what we send actually gets to the printer. More must be noted before we can be confident of that.

Before printers became so educated my programs used to halt and give the operator advice when they ran out of paper. In this case, or in the event someone accidentally kicked the plug out of the wall, my program would wait patiently until told to carry on. That is the way we did it a long time ago, back around 1982, or so.

In fact, an old (1986) Big Blue manual says printing is asynchronous with processing. That is so much bull. It was bad advice then, it is now, and it will likely be wrong from now on.

BASIC still thinks that what is happening is asynchronous. It sends bytes to the adapter. The adapter sends them to the printer, and the interpreter monitors the adapter to see if the printer sends back a problem signal. The time interval between your PRINT and what is happening on paper defies the word asynchronous. Big buffers is why.

A little six pound two hundred dollar printer has a whopping 8,000-byte buffer. Enough memory to hold the print-image for six or eight paychecks, maybe. Imagine the problem of, when

the stack of blank checks runs out, five checks before the end of the payroll. Nobody knows it.

The printer's cry for help will probably not be heard. You sent the check lines out as fast as you could, trusting that they would all get printed. Then you closed up shop and returned to your start-up menu, or worse yet, branched to the routine that does a batch update of your master file.

There are numerous error codes in BASIC that can be triggered by printer related statements. Any one code would suffice. It really makes no difference to me what is wrong. My response is invariably the same: Abort. Noisily, of course.

Most of the time a BEEP is not enough. It would be helpful to your operator, and both of us, if the manuals were clearer on how to discern just what is wrong. Here are some things learned the hard way.

If you do an OPEN and get ERR = 68, "Device unavailable", it is because there is no adapter in the machine for that printer port (or the adapter is kaput).

On the other hand, if a working adapter does exist, no error is triggered on OPEN, even if the printer is off, off-line, or it has been stolen. (Contrast this with an OPEN to a data file; if it doesn't exist, but should, you get an immediate error.)

If you do LPRINT or LLIST to a non-existing or non-working adapter you will get ERR = 57, "Device I/O Error". Maybe.

If you do WIDTH, as in WIDTH "LPT1:", for example, no error occurs no matter what exists, does not exist, or is not working.

Having succeeded in doing WIDTH, which always succeeds if everything is spelled correctly, if you then do an OPEN, an LPRINT, or LLIST to an absent or lazy adapter, it seems the interpreter changes its mind because it will then generate ERR = 55, "File already open". Which is crazy. Or at least, illogical.

Three other error codes are possible when one variant or another of PRINT is done. If it fails. When you read the manuals see why I consider error codes 24, 26 and 27 as if

they were all alike. Once upon a time it was nice to be able to react differently to each. Today it is a blamed nuisance that there are three. As it has always been a nuisance that a simple range test was not an exact technique, because that odd error code (25) has nothing to do with I/O.

A well-designed application has to anticipate that many kinds of failures can occur while printing, and provide some means to do a restart from a midpoint, or, allow for a complete rerun without double whacking already updated files. When a printer does cry for help, there is no way to correlate what you just sent with whatever it just got around to trying to print. And, postoperative cries for help will fall on deaf ears.

Even though software and hardware technologies are advancing rapidly, but not always at the same pace, it would seem that the manuals writers could give us some indication as to what is contemporary and what is archaic; what is useful and what is not; what works and what doesn't.

Hopefully this chapter has provided some of that, and a bit more, for those three normal devices: the keyboard, monitor, and printer. What is abnormal in my book, still, is things like a mouse, a light pen, and making music. Fun, yes, but irrelevant to the business of doing serious data processing. We can do that, but we have to be more clever than ever, to do it in BASIC, on machines that are more adroit at playing games than producing profit for their owners, or you, or me.

Chapter 7 = GRAPHICS

Graphics can be fun. Especially so, in the context of a game. A game, in the sense of, an intellectual challenge: Can you learn to do it armed with nothing more than a machine with the mechanical capability, BASIC language manuals, operating system manuals, and hardware reference books.

Be prepared for a real contest. My game has not yet ended. It likely never will. At this point we have reached a stalemate. My unnamed opponents (that wrote the manuals) won a couple of rounds. What follows are things learned from the rounds I won, before my stamina began to wane, and my budget gave out.

To write a technical work that would accurately cover all of the different types of adapters and monitors would require a lot of hands-on testing. And a lot of money. And the book would never get printed because new hardware of this genre is born everyday, so it seems.

What is included here covers MDA, CGA, and EGA. The Monochrome Display Adapters have been around the longest (1981). Color Graphics Adapters were the first machines that provided monitor output in color (1981 also). Enhanced Graphics Adapters were next, offering more colors and higher resolution images (1985).

VGA--Video Graphics Array--is so new it is not yet named in the BASIC manuals (1987). The Hercules Graphics Card (1982) is not identified, as such, either.

Thus far my experience has been that what worked on the older hardware works on the new. To take full advantage of the new, or the old, requires technical knowledge of a specific piece of hardware, and a basic understanding of how BASIC talks to any of them.

This chapter has two goals: To bridge the gaps between the manuals, and to provide some freeze-dried code that is useful. The manuals are all written so as to stand alone; they seldom allude to each other. The coding examples in the BASIC manuals are meant to convey concepts; they are rarely usable as shown.

A logical first move for this game is to state what is meant: In BASIC, any SCREEN statement whose first argument is other than zero puts you in "graphics" mode. Conversely, SCREEN 0

is "text" mode.

One fundamental difference between these two modes is in how printable characters are stored in video memory, and where they come from. So, a second move in this game should be to review where PRINT gets its characters from, for both, text and graphics modes.

The upper range of characters in the adapter's font set--CHR$(128) to CHR$(255)--are sometimes called pseudo-graphics characters. Just why "pseudo" is used, is beyond me. Agreed, they are non-ASCII. They are just as useful when in graphics mode as they are in text mode, however.

This is equally true of those low-ball characters that are in ASCII range (0-127), that are pretty, but not, "pure" ASCII. That standard says codes below 32 are control codes and their exact meaning can be further defined by a manufacturer. Thus, those codes have been further defined as printable characters. Sometimes. Depending on....

All of the characters that can be shown on a monitor attached to a monochrome adapter come from dot patterns burned into ROM on the adapter board. That font set covers the full range of 256 character codes (0-255).

This is true for the graphics adapters, too, when in text mode. When an adapter is commanded to switch to graphics mode, the character set, or part of it, is "soft"; the dot patterns for some characters may come from RAM, rather than from ROM.

> Distinction: RAM--short for Random Access Memory--can be changed via software. ROM--short for Read Only Memory--cannot be changed. While ROM can be read, "randomly", what is stored in it is permanent; it was burned-in (as software) electronically, by the manufacturer. Thus, bits in ROM are frozen, as is, with or without power.

Which brings up the issue of GRAFTABL.COM: A "soft" character set that is needed on CGA machines if you want to PRINT or use the function-version of SCREEN, for character codes that are in the 128-255 range. The more expensive graphics adapters have this set of characters stashed in ROM, also, in addition to the lower-128 set of codes. If you load a soft-set of characters, however, such as in GRAFTABL, all graphics adapters quit using their "default" font set for codes 128-255, in favor of the soft ones, when operating in graphics mode. (In text mode they

use the standard 0-255 font set, no matter what has been loaded by GRAFTABL.)

Now, for the next move: In text mode characters are stored in video memory as codes. Adapter electronics continuously scans the monitor and that memory. Each print position on the screen correlates to a specific byte in memory. As those bytes are scanned, a character-generator chip on the adapter repeatedly converts the codes into corresponding pixel (picture element) patterns.

When in graphics mode the character generator chip is bypassed. What are shown on the screen then, are pixels that correlate to bits in video memory. In concept this is simple; if a bit is set on (a binary one), a dot on the screen is illuminated. If a bit is off (a binary zero), a blank spot occurs.

Which bits (of certain bytes) align with a given dot-position on the screen can wait a couple of pages. At this point note that, when we PRINT a character in graphics mode, the code that would normally be stored in memory is immediately decoded, and the character is stored as individual bits, rather than as a 1-byte code number.

Thus: Graphics mode characters are "soft"; they are made up of dots (bits on or off). Text mode characters are also shown as dots, but they are stored as codes, and are repeatedly decoded by adapter electronics. Knowing this basic difference, see how PRINT produces a "canned image" of pixels, but, after they are printed (decoded) in graphics mode, they are stored in memory at that point as primitive bit patterns.

When we program in graphics mode the commands we use also create bit patterns in video memory. This may be done on the basis of one pixel at time, or with some commands, multiple sequences of pixels. Which is how we create images like boxes and circles. In concept, see that PRINT is really no different than other "macro statements" that generate complete "icons".

A new DOS-native capability was added in Release 3.3: DLL, i.e., Down-Line-Loadable code sets for printers or monitor adapters (or any other device that plays by the same rules.)

This new feature of DOS is generically called "Code Pages". It was conceived primarily for use with foreign languages. It is

complicated, however, and the manual's descriptions of how to make good use of the capability is so confusing that if you are so inclined, reserve a hefty chunk of court time; it is going to be a long tournament if you are determined to win.

The major difference between devices that support DLL Code Pages (viz, EGA/VGA), and those that do not, is that the entire range of codes (0-255) can be overwritten by software supplied bit patterns. They can also be changed on the fly, and used in both text and graphics modes. On the fly meaning from DOS, not from BASIC. Meaning, before loading BASIC, or via SHELL.

As is always the case, SHELL is risky. Read the definition of SHELL in the BASIC manuals with one eye closed. GRAFTABL, for example, is a single-shot DOS command. Only parents can do it. Do not try it using SHELL. Run as a child process, GRAFTABL has no respect for its parents and will likely hang the whole family. It will not let you return to BASIC; it may even force a manual reboot. Once done, successfully, what is loaded is what you live with until the next reboot. There is no means for getting rid of it, and a second "load" will not work to overlay a previously loaded font set.

At about this point some books would refer you to an appendix to see what the printable characters look like, and which codes are assigned to each. Forget it, if you want to know for sure.

Charts in hardware manuals can be trusted more so than those in software manuals but, it is not unusual to find differences in what any of the books show, and what characters really do look like. The following tiny BASIC program can be trusted to tell the truth, on any machine, at any given point in time, for the text-mode set of characters then in residence.

```
10 CLS:LOCATE 16,1          'clear and move the cursor
20 DEF SEG = &HB800         'for mono use &HB000
30 B = 0                    '1st offset into video RAM
40 FOR I = 0 TO 255         '0-255 character codes
50  POKE B,I:B = B+4        'show one:bump offset by 4
60 NEXT
```

Take note that, in the above routine POKE puts codes directly into memory. BASIC has no idea why you are doing it, so it does not attempt to translate your intentions. POKE was used because BASIC will not PRINT all of the characters possible. It will print them all save for the following codes when used

in a CHR$ statement (for example):

decimal	ASCII meaning	BASIC behavior
7	BEL - alarm	BEEP, print nothing
9	HT - horizontal tab	prints 7 spaces
10	LF - line feed	down 1 line, position 1
11	VT - vertical tab	home (same as LOCATE 1,1)
12	FF - form feed	clear (same as CLS)
13	CR - carriage return	works same as code 10
28	FS - field separator	reposition 1 space right
29	GS - group separator	reposition 1 space left
30	RS - record separator	up 1 line, same column
31	US - unit separator	down 1 line, same column

What is happening here is, BASIC "translates" what you say to print--as in PRINT CHR$(9)--and does function calls to the BIOS, or otherwise commands the adapter, rather than transmit a character code, per se.

BIOS: Basic Input/Output System, which is boot-ROM and DOS related. The acronym BASIC has no kinship to the BIOS's first name. Also: In the new PS/2 machines there is an ABIOS and a CBIOS, to support OS/2 and the old DOS. As used here, BIOS is a handy name for them all.

Add to this wealth of trivia, the BIOS for video output may be in one of three places. For MDA and CGA it is an integral part of the system BIOS, i.e. it is on the mother board. For EGA and VGA, all interrupts for video services are rerouted to a specialized set of BIOS routines contained in the adapter. The Hercules-like adapters (and some other newer types) depend on software drivers. On boot up these device drivers are loaded into memory, and the interrupt address for video services is modified to point to these product-specific drivers.

How to spell all of these acronyms correctly, and who invented them when, is somewhat trivial. Programming in BASIC, the rest is trivial also when we depend on the interpreter to do what we ask it to. It does provide an easy, dependable, interface to all of these low-life mechanics. An intellectual awareness of what is happening down below, however, can make it easier to see what we are seeing. On the screen, and in the manuals.

Most of the above is really basic, including that about BASIC. In text mode, all characters are stored in video RAM as 1-byte

codes (0-255), each one followed by an attribute-byte. How to address those bytes is covered in Chapter 6. Graphics adapters offer more capability than monochrome. Staying with text mode, for the moment, here are some new opportunities.

WIDTH 40: Character fonts are the same as WIDTH 80, but are two print columns wide, per character. On the screen. They still use 1-byte of storage per, so a line on the screen is 40 columns; a line in video RAM is 80 bytes.

PCOPY: The fastest gun in the west when it comes to having to "switch" screens. (BSAVE and BLOAD are still useful, and are described in more detail, later. PCOPY is a much better alternative for some screens.)

MEMORY: Video RAM is hardware mapped. Regular software does not load into this memory space, so it is a safe place to "put things of your own", without worrying about corrupting other software that may be in memory. (Unless, of course, somebody else uses this space in a similarly uncouth manner.)

How much memory is an interesting question. The business of using PCOPY, for example, requires that you know this little fact. Suppose your machine was a flea market special, and it came with no documentation. (Or, it cost a small fortune, but you have no confidence in the documentation, or your ability to count the chips accurately.) There is a relatively painless way to get the machine to help you. PCOPY itself can be used to help determine the amount of video RAM available.

Working in text mode, WIDTH 80, each full-screen page requires 4kb. So, if PCOPY 0,3 works, but PCOPY 0,4 gives an "Illegal function call", the adapter has 16kb. That is, 4 pages of 4kb each; the pages are numbered 0, 1, 2, and 3.

In SCREEN-0-WIDTH-40 mode, the number of PCOPY pages doubles because each full "page" uses only 2,000 bytes. (Remember that PCOPY is always illegal on a monochrome adapter. BASIC manuals say that PCOPY works in all screen modes; another instance that belies their use of the word "all".)

Presumably you know what kind of adapter is in your machine. To be able to write programs that are self-adapting, in the event they are supposed to run on any machine, it may also be necessary to determine which graphics adapter is in use. (If

SCREEN 0:PCOPY 0,1 fails, the issue is mute: Monochrome.)

One tell-tale difference between CGA and EGA is in how memory is used by the adapter. See this, assuming PCOPY 0,3 works:

```
10 SCREEN 0:CLS
20 DEF SEG = &HB800
30 POKE 16384,65
RUN
Ok
```

If what is shown in the upper left corner of the screen is "Ak" rather than "Ok", the adapter is CGA and not one of its fancier successors. This example is useful to demonstrate the need for technical awareness when we opt to "go around" the interpreter.

Video RAM in a CGA begins at address &HB800 and consists of a full 16kb, up to, and including address &HF7FF. Using decimal numbers, 16kb is 16384, and the address range is 47104 through 63487. A not so trivial piece of trivia for a CGA is the next 16kb. It too can be addressed, with POKE for example, but it is phantom memory. The adapter "wraps" this second 16kb range of addresses to match the base address that begins at &HB800. (Which accounts for the "Ak" seen above.)

EGA, on the other hand, has an honest 64kb of memory, and VGA has 256kb, addressed as 4 blocks of 64kb, each. At least. Not all of this memory is video RAM, however. Part of this address space contains the video services BIOS, mostly in ROM chips.

The above memory figures are honest assumptions. Yours, mine, and the BIOS routines. What actually exists, in chips plugged into the adapter, may vary. PEEK and POKE can be used to find out the truth, if the truth need be known, which is needed, if you want to use PEEK and POKE. Even so, be watchful of lies, like in the case of the CGA's tricks with memory addresses.

An easy way to determine CGA vs. EGA, and to determine which type of monitor an EGA has been switch-selected for, is to experiment with different SCREEN statements. Any SCREEN mode argument larger than 2 will cause an error on a CGA (ERR = 5). SCREEN 7 and SCREEN 8 are valid--will not cause an "Illegal function call"--on any of the more sophisticated adapters. SCREEN 9 and SCREEN 10 are usable with an EGA adapter, but only if the option switches on the adapter are set for use with an "Enhanced Display" monitor (as opposed to a "Color Display",

e.g. RGB: Red, Green, Blue).

Some texts would encourage you to use operating system "service calls" to find this out. My habitual inclination is to not go native unless there is no practical alternative.

One reason for this attitude is, you cannot always trust the answers you get by sampling BIOS bytes. It can take umpteen hours to determine their truthfulness. And truth is relative. Let BASIC make its own determinations. Even when it decides incorrectly, we are bound to abide by its judgment, anyway. (If we want to use the natural capabilities of the language.)

Which technique to use, and when, depends on your confidence in the configuration data stored in the BIOS. Which is where BASIC gets its advice from. All of which depends on the "tech" that installed the boards, and whether or not he set the little switches and jumpers correctly. And whether or not he who made that board followed the "standard" rules about switch meanings. And on, and on, and on.

Now, on with the show: How to show what you want to see. In BASIC. An opening move for this phase of the game is to see a list of the key words that are peculiar to doing graphics, and those that are useful otherwise, but behave peculiarly in this mode.

The following list of key words are those invented especially for graphics. These are the ones that will generate run time errors (ERR = 5, "Illegal function call") if SCREEN 0 is in force at the time they are encountered.

CIRCLE	POINT
DRAW	PRESET
PAINT	PSET
PALETTE	VIEW
PMAP	WINDOW

Two key words that existed before graphics was invented can now be used in a different context: GET and PUT. They date back to the early days of disks. Now, while in a SCREEN mode of other than zero, they can be used to exchange blocks of bytes between the adapter's memory, and numeric arrays in your own program. Which, as we will soon explore, can be more fun than a chicken plucking contest in June, any time.

Two of the key words that were invented when graphics made its debut are useful in both text and graphics modes (and with a monochrome adapter): COLOR and SCREEN.

Four old key words--LINE, PRINT, STEP, and USING--now play new roles when used in combination with the key words that were invented just for graphics.

Two of the oldest words in BASIC are WIDTH and CLS. They were in MBASIC, even, back in the seventies, in the realm of CP/M. In those days the manuals described them accurately. Their meaning and behavior has been modified so many times since, it is no small wonder that the people who write the manuals cannot precisely describe what they do, how they work, or when to use them.

Here are some notes from the margins of my manuals. Thankfully they leave us a lot of white space for such jottings.

CLS: In text mode, all character bytes are set to CHR$(32); all attribute bytes are set according to the second argument of the most recent COLOR statement. My newest manual refers to "alpha mode"; it must mean text mode (or maybe a beta mode is coming). CTRL-HOME clears the same as CLS, but that only works if (when) the program has halted for keyboard input. If function-key legends are being displayed, line 25 is also cleared in memory, but it is then immediately restored to what is being shown; both the text, and their companion attribute bytes.

WIDTH (GW-BASIC): Does nothing if the argument used (40 or 80) is the same as what is already in force. An easy test for determining the current setting is to LOCATE 1,41. This will error trap (ERR = 5) if the current width is 40; if no error, current width is 80. Do not use the WIDTH "SCRN:" form of syntax. If in 40-column mode and an illegal argument is used--not 40 or 80, as a literal, or from a variable--the interpreter gets confused. From then on it thinks 80 is 40, and vice versa. Which can really drive you up a tree. (The only way out of this mess, for sure, is to quit and go back to DOS and reload the interpreter.) Now, see this:

```
1000 CLS : DEF SEG = &HB800  'prepare to load a BSAVE file
1010 BLOAD "PICTURE.MSK",0   'a graphics screen "mask"
1020 WIDTH 80                'ready to print text
```

The "note" in the manual is wrong. Do not expect WIDTH to

act like CLS. If the above program is jump started--as in GWBASIC <program>--the screen will be blank and the PCOPY page-0 will be "clear". If you BREAK and do a RUN--as we do when debugging--nothing gets cleared. (This is also true if line 1020 had said WIDTH 40.) The moral to this story is: Always issue a SCREEN-WIDTH-CLS sequence if you want to be sure of what will be seen, and, what the not yet visible parts of video memory looks like. PS: Always do it before a BLOAD, if what is being loaded was not saved in the current mode.

WIDTH (QuickBASIC): All bets are off about the 40 vs. 80 business. The compiler permits WIDTH 40 or WIDTH 80, and the result is "compatible" with the interpreter. It goes beyond the interpreter, however, and permits 0-255 to be used as an argument for WIDTH "SCRN:"--although, if WIDTH "SCRN:",0 is encountered, an error will be provoked at run time. There is no easy trick that can be used to find out what WIDTH is in vogue while a compiled program is running, so it is up to you to keep track of it yourself.

PCOPY: The manuals say to see CLEAR for more information. Which is interesting reading material, but it has nothing to do with PCOPY. (An old typo most likely; probably should say, see CLS.) When editing program lines, DO NOT use PCOPY in immediate execution mode. Especially on an EGA. The small variations in video services between the system BIOS and the one on an EGA can sometimes cause the BASIC editor to garble your program.

VIEW PRINT: Don't. Not if you want to LOCATE on line 25, and the program is supposed to run on anybody's machine. In the event you establish a text window, then later attempt to revert to using the whole screen, a LOCATE to line 25 may cause an error (ERR = 5). This is probably related to BIOS variations, also. My only known cure, if this happens, is to dump everything and reload the interpreter.

"Viewport": This term is not defined, as such, anywhere. We are supposed to infer its meaning by reading the separate pages on CLS, VIEW, WINDOW, and their kin. Because they are inconsistent in terminology, and because this "concept" does not produce consistent results among the different adapters (various BIOS implementations), and because the interpreter and compiler behave with slight variations, the best advice is: Don't rock the boat. In a given machine, working with a given release of software, this concept can be appealing.

It is not unlike atomic isotopes, however. The half-life of a given program may be dramatically different, depending on the numbers involved in the nucleus in which it was written.

SCREEN (function): The newer manuals say this only works in "alpha mode". (The older books did not warn us this may not work in graphics modes.) The truth, in any event, depends on the IQ of the BIOS that is providing video services. Because characters become just so many bits in graphics modes, to respond to a BASIC request for a character's code the BIOS has to unscramble the bits to see what character they match. Some can do it, and some cannot. Also: In a CGA, which can re-encode characters from pixel patterns, it can do it for codes 128-255 only if GRAFTABL is in residence. In all cases, it is critical that matching font sets are involved. A BSAVE image, for example, can be interpreted correctly only if the character set then in residence exactly matches the one used at the time the image was generated. Which means you have to be very careful when writing programs on one machine that are destined for another. (This applies to entire character sets now that DLL Code Pages lets anyone fiddle with the fonts.)

SCREEN (statement): The books all say the page arguments are usable only with EGA. Nothing prevents you from doing the same thing on a CGA, effectively, using POKE and PCOPY (for text), but it does take some programming work, and poking verbose text takes a lot of execution time. A real effective trick is to BLOAD text pages directly into video memory slots that correspond to PCOPY page numbers.

So goes my home remedies for the key words that give the worst headaches. One more note is sorely needed, but there is no handy place in the manual to write it. Except maybe, up front. One of the meanest lessons to learn the hard way is to not edit a program while in any graphics mode. While debugging graphics programs it can be a nuisance to have to keep switching back to text mode, but there is a real risk that the editor will garble a program that is modified while in any of the graphics modes. (This relates to my earlier notes about PCOPY, by the way.)

Now for some free code out of my modules library. These BASIC routines are useful to show how some of the graphics statements can be used. They are not textbook examples. They are bona fide chunks of code from products that have been delivered to live clients.

The first two modules are useful for enlarging images; the next two are for rotating objects about their own axis. All four of these use POINT to "read" pixels already on the screen and PSET to "write" pixels where they are wanted. Because they conform to conventional programming practices, they are equally useful in all graphics modes, and work with either the interpreter or the compiler.

From now on assume DEFINT C-L. While not critical, performance is better with integer variables. With the interpreter, they are all faster if remarks are omitted, and logic is condensed into multiline statements. (Chapter 10 shows how coding style impacts interpreted programs, and how to choose techniques that give the best possible performance.)

To double the width of an image:

```
1000 SCREEN 2 : DIM D(15)   'dot tank
1010 LOCATE 2,2:PRINT "W"   'an 8 x 8 test icon
1020 FOR E = 8 TO 15        'row loop
1030  FOR F = 8 TO 15       'pick-up loop
1040   D(F) = POINT(F,E)    'save dots on 1 row
1050  NEXT : C = 8          'C = first column
1060  FOR F = 8 TO 15       'replot 1 row
1070   PSET(C,E),D(F)       'put 1 dot
1080   PSET(C+1,E),D(F)     'second put, same dot
1090   C=C+2                'bump column count
1100  NEXT                  'finish 1 row
1110 NEXT                   'finish all rows
```

To double the height of an image:

```
2000 SCREEN 2 : DIM D(15)   'dot tank
2010 LOCATE 2,2:PRINT "H"   'an 8 x 8 test icon
2020 FOR E = 8 TO 15        'columns loop
2030  FOR F = 8 TO 15       'pick-up loop
2040   D(F) = POINT(E,F)    'save dots in 1 column
2050  NEXT : L = 8          'L = first row
2060  FOR F = 8 TO 15       're-plot 1 column
2070   PSET(E,L),D(F)       'put 1 dot
2080   PSET(E,L+1),D(F)     'second put, same dot
2090   L = L+2              'bump row count
2100  NEXT                  'finish 1 column
2110 NEXT                   'finish all columns
```

To invert (flip) an image on its horizontal axis:

```
  3000 SCREEN 1 : WIDTH 80
  3010 LOCATE 2,2 : PRINT "L" 'an 8 x 8 test icon
  3020 FOR H = 8 TO 15        'columns loop
  3030  I = 8 : E = 15       'I = 1st row : E = end row
  3040  WHILE I<E            'from 1st to end
  3050   J=POINT(H,I)        'save dot at H,I
  3060   PSET(H,I),POINT(H,E) 'replace H,I with dot from H,E
  3070   PSET(H,E),J          'replace H,E with saved dot
  3080   I = I+1 : E = E-1    'down 1 row : end = 1 less
  3090  WEND                 'finish 1 vertical line
  3100 NEXT                  'finish 1 horizontal line
```

To reverse (spin) an image on its vertical axis:

```
  4000 SCREEN 1 : WIDTH 80
  4010 LOCATE 2,2 : PRINT "C" 'an 8 x 8 test icon
  4020 FOR H = 8 TO 15        'rows loop
  4030  I = 8 : E = 15       'I = 1st col : E = end col
  4040  WHILE I<E            'from 1st to end
  4050   J=POINT(I,H)        'save dot at I,H
  4060   PSET(I,H),POINT(E,H) 'replace I,H with dot from E,H
  4070   PSET(E,H),J          'replace E,H with saved dot
  4080   I = I+1 : E = E-1    'right 1 col : left = 1 less
  4090  WEND                 'finish 1 horizontal line
  4100 NEXT                  'finish 1 vertical line
```

An interesting trait of POINT and PSET (and PRESET) is that they can be vectored into thin air. Even so, as the books say, their arguments must stay within the natural range of integers (-32768 to 32767). In the above routines the coordinates that are shown align with the positioning mechanics of LOCATE, i.e. uniform spacing in increments of 8 columns (16 if WIDTH 40) and 8 rows. This does not have to be, of course; the logic itself can be used for any size of rectangular icon, placed anywhere, by changing the start and stop arguments for the loops.

This next routine uses DRAW, CIRCLE, and PAINT to draw a big clock in the middle of the screen, in "medium resolution". My term. Various texts play with this. Some used to call this "high resolution". When higher resolution images came along, medium, high, and very high resolution became state of the art

terminology. (Interestingly enough, "low resolution" is not a popular term.)

Here, in SCREEN 1 or SCREEN 2, with WIDTH 40, this shorty uses graphics statements, and LOCATE and PRINT. And DRAW, which is a handy way to dangle the angle of clock hands.

```
  5000 DEFINT C-L:DEFSTR M-Z                    'define types
  5010 T5=SPACE$(5)                             'time$ tank
  5020 SCREEN 1,0:WIDTH 40:CLS                  'screen set up
  5030 LSET T5=TIME$                            'get system time
  5040 L = 360-(30*VAL(T5)+VAL(MID$(T5,4))/2) 'long hand angle
  5050 H = 360-(6*VAL(MID$(T5,4)))              'hour hand angle
  5060 CIRCLE(160,100),7,2                      'dot at junction
  5070  PAINT(160,100),2
  5080 DRAW "C2 TA=L; NU36"                     'draw long hand
  5090 DRAW "TA=H; NU50"                        'draw hour hand
  5100 FOR I = 86 TO 90
  5110  CIRCLE(160,100),I,2                     'rim around clock
  5120 NEXT
  5130 LOCATE  6,19 : PRINT 12 : LOCATE 20,20 : PRINT 6
  5140 LOCATE 13,11 : PRINT 9  : LOCATE 13,28 : PRINT 3
  5150 CIRCLE(198,52),1 : CIRCLE(198,150),1   'dot @ 1 & 5
  5160 CIRCLE(222,72),1 : CIRCLE(222,126),1   'dot @ 2 & 4
  5170 CIRCLE(122,52),1 : CIRCLE(122,150),1   'dot @ 7 & 11
  5180 CIRCLE(98,72),1  : CIRCLE(98,126),1    'dot @ 8 & 10
```

As used here, DRAW is handy. It is used to draw two short lines. Which could be done with LINE. In this case, it also takes care of the angle of those lines. Which is one of the few good uses of the "Graphics Macro Language".

Examples of DRAW in the older manuals use string concatenation. For other than observation, such examples are useless. (See Chapter 4; it dwells at length on why concatenation should not be used in real programs.) The alternative is to set up a base set of "command strings", then use MID$-type mechanics to alter them during program execution, or to code one whale of a lot of quoted strings. In the end, because of all of the overhead needed to avoid memory fragmentation, you too may conclude: DRAW is handy, but not as often as its advocates would lead us to believe. More personal cynicism: Graphics programming is highly humdrum, no matter how you do it.

In the routine above, see the tedium involved in plotting the dots that are used in place of numbers on the clock face. In fact, this short program took over eight man-hours to develop.

Which included the time to design, code, debug, and test, with CGA, EGA, and VGA, for both SCREEN 1 and 2, using interpreted BASIC. (DRAW in the early releases of the QuickBASIC compiler could not include the syntax used here, forcing us to use the cumbersome VARPTR alternative. And that is very humdrum.)

In the interest of saving time, which is of prime interest to those of us programming for a living, the need for a graphics tool soon becomes obvious. My own, coded entirely in BASIC, is not unlike a lot of the "paint" programs in the stores. (In fact, it was development of that tool that first gave birth to the early routines now in my modules library, including those above that can enlarge and rotate icons.)

The big advantage to a homemade tool, written in BASIC, is the ability to generate "cut and paste" images that can be stored in an icons-library; images that will be "compatible" with any end-product application program also written in BASIC.

Two different techniques can be used. BSAVE and BLOAD are best for full screen "masks". They impose no memory burden for the end-use program. PUT and GET are best used for partial-screen images, and for assembling full page montages to be stored as a final BSAVE file. But PUT and GET do require vast chunks of temporary storage, for both, the generating program and the one that needs to merely do a GET so that it can then PUT what was gotten.

The easiest of these two techniques to implement is BSAVE and BLOAD. All that need be known, basically, is how much memory to save. PUT and GET demands a tad more than can be gleaned from the manuals. Taking the easiest first, see that BSAVE and BLOAD of graphics screens is similar to doing it in text mode. (Chapter 6 gives that detail.)

Graphics pages are mapped in 16kb blocks. In SCREEN 1 or 2, for example, the size-to-save argument is 16384, which is the inclusive total of from 0 to 16383, counting position-0 as 1.

Thus: BSAVE "picture.msk",0,16384 stores an entire screen.

Then: BLOAD "picture.msk",0 will reload (redisplay) it.

Of course, DEF SEG = &HB800 must be issued first, in either of these cases. As a matter of habit, do a default DEF SEG as soon as possible after whatever it was that required that it be

changed in the first place. That advice is in the fine print in the manuals, but they do not say why. My advice is based on an awareness that the interpreter can run amok once in a while because it uses the "default address" itself. It is not easy to get a handle on just when, however.

One rarely documented piece of trivia affords an opportunity when doing a BSAVE of 16kb graphics screens. There is a hidden string of 192 bytes that are not used for anything, but they are saved in a BSAVE file, and therefore, reloaded by BLOAD.

The advantage: COLOR, per se, is not saved with BSAVE, nor for that matter, is the 1 or the 2 used for a SCREEN mode, or whether WIDTH 40 or 80 was in effect at that time. So, POKE can be used to store these values in a file that goes to disk, so that a using program can automatically conform to whatever adapter settings existed when the file was generated.

This can be done in one of two ways: Do the BLOAD then use PEEK to condition COLOR and WIDTH statements, etc., or by opening the picture-file first, as a relative file, so that a GET can be done to obtain the values needed to condition the adapter prior to doing BLOAD.

The (almost) secret 192 bytes are located between the two 8000 byte blocks that make up a 16kb graphics page. The first byte of that free space can be seen by: DEF SEG = &HB800, then a PEEK(8000). To discern the magic of 192, remember that 8kb is really 8192, and two times 8192 is 16384, which is the sizing argument needed to BSAVE all of a 16kb graphics page.

Here is how to see the beginning of that "hidden" string of bytes before a BLOAD is done:

```
OPEN "picture.msk" AS 1 LEN = 1    'relative file
FIELD 1,1 AS X                     'DEFSTR X already done
GET 1,8008                         '1st byte is now in X
```

The record (byte) pointer is determined by adding 8 to 8000, to account for the 7-byte header in a BSAVE file. (What the header bytes contain is covered in detail in Chapter 6, also.)

When opting to go this route, be sure to CLOSE a file that is about to be implicitly opened by BLOAD, or you will trigger a "File already open" error. PS: BSAVE and BLOAD both cause an implicit OPEN and CLOSE sequence; another little omission in the language manuals that is irritating to have to learn by

trial and error.

As we have come to expect, trial and error is the only sure way to find out the full truth about the use of PUT and GET, also. Here are a few more notes from the margins of my manuals; they might save you the aggravation of some of my errors.

To GET an image from video memory into an array, the name of the array must be specified in the GET statement. Ok. They forgot to tell us that the array must have been explicitly dimensioned beforehand. (Traditionally, in BASIC, DIM only had to be specified if any of the dimensions was larger than ten.)

PUT cannot be used if no GET has yet been done. At a glance it would seem that one would only want to PUT after a GET. In practice, however, it is easier to code routines that are supposed to PUT what used to be, before doing a GET at a new location. To make this possible, immediately after a DIM of a pixel tank, force-load dummy "image dimensions" into the array. This can be done in one of two ways:

```
DIM D(100) : GET(0,0)-(0,0)

or

DIM D(100) : D(0) = 1 : D(1) = 1
```

The second trick shown here is the equivalent of what would have been the natural consequence of the first one, assuming that DEFINT D had already been done. The advantage of the the first method is that a first PUT will reproduce whatever already exists at that initial location. (Assuming PSET is the "action verb" following PUT.)

The older manuals (1986) did not give us a clue about how to compute how large an array had to be, to hold an image of a given rectangular dimension. This oversight forced me to do a lot of probe coding; "Illegal function call" is not very instructive.

The newer manuals provide a formulae, which is nice, and it works. One more bullet would have been better still, to keep you from getting shot down inadvertently. The formulae shown requires an awareness of how many pixels are used in various SCREEN modes. When you follow their cross reference to the

description of SCREEN to find this out, you still may come up short. WIDTH 40 requires twice as much array space as does WIDTH 80, in any of the graphics modes. In fact, it would be nice if they went all the way, and explained that graphics pixel-mapping is established by the combination of SCREEN and WIDTH, not necessarily by a SCREEN statement alone.

The formulae shown in one of the BASIC manuals is echoed here, in the event yours does not have it. Use this (if WIDTH 80, divide this by 2) as the DIM-size for the smallest possible integer array:

```
4+INT(C*I+7)/8)*L
```

where C = columns, I = bits-per-pixel, and L = lines (rows).

It is equally nice that the description of PUT warns us that an "Illegal function call" will occur if the image to be PUT is too large to fit on the screen. Because the error-trap mechanics in BASIC are somewhat flaky (read Chapter 9), it is better to know what is legal in the first place. Then you can decide whether or not you want to take a chance on the sheriff catching you.

Sticking with integer-arrays--which is practical because there is no advantage to using floating point arrays--the first two array elements contain the column-count, and the row-count, of an image captured with GET. By testing these two values just before a PUT, we can determine if we are apt to go over the edge and bring down the long arm of the law.

Assuming OPTION BASE 0--which is a safe assumption because BASE 1 has never been seen in any of my programs--see this:

```
GET(0,0)-(C,L),D       'D(0) = C+1 and D(1) = L+1
```

These are pixel position counts, e.g., if D(0) is ten, add 9 to the first argument in a PUT to see if there is enough room for the width of an image. In the same way, if D(1) is five, add 4 to the second argument of a PUT to see if the length of an image will fit. PS: Maintain an awareness also, that the C-factor (columns) is the one that is double for WIDTH 40 vs. WIDTH 80. The value in D(0) is the actual number needed if no change in width occurs between the GET and PUT.

Now that our manuals have been illuminated, we can get on with

the business of creating icon files. Which is fairly simple: BSAVE "icon.msk",F,L--none of which has anything to do with graphics, because BSAVE can copy any chunk of memory into a "binary file" on disk.

Continuing with the same names in the above one-liners, the value for F--the "from address"--is equal to VARPTR(D(0)). Not to beat a dead horse, but notice that the default DEF SEG is the correct one to be riding before a BSAVE, for anything coming from an interpreted BASIC program's own working storage area. (For LNA addresses--"Large Numeric Arrays"--a term invented by the compiler writers, see the QuickBASIC manuals.)

The L-for-Length factor, the number of bytes to be saved from an array, must be at least as large as the GET-image required. It does not have to be as large as the array itself, of course, in the event you are using a large, general purpose dot-tank to hold images of various sizes. Again using the preceding names of variables, here is my formula for determining L, which is a minor variation of the one used for calculating the size of an array needed for GET:

```
4+INT((D(0)+7)/8)*D(1)
```

The difference being, obviously, the bits-per-pixel factor is omitted here, since it has already been accounted for.

The BLOAD counterpart to BSAVE, for home-brew icons, is rather simple too. Use VARTPTR(D(0)) as the offset argument in BLOAD, and do a PUT of array D. To preclude an undesirable run time error, the same test suggested earlier can be used to ascertain whether the image to be output will have enough room toward the right, and downward, from the vector named in PUT.

There are times, here too, that it would be nice to know what graphics mode was in use at the time an array-icon was saved. An easy way to make an icon file self-identifying is to store a few pointers in the array, just before a BSAVE is done.

As a convention, for example, to flag WIDTH 40 vs. 80, set D(0) to a minus, and tell everyone that when they first BLOAD an icon, to ascertain the sign of D(0), then immediately force it to be positive; like D(0) = ABS(D(0)). (This is essential before doing any PUT.)

In tight corners, to make sure that a dot-tank array is large

enough to hold an anticipated icon file, the size of the file itself can be examined. Like this:

```
OPEN "icon.msk" as 1 : B = LOF(1) : CLOSE 1
```

The smallest integer array that can hold this file will be one with a DIM not less than: (B-8)/2. This is valid for newer releases of BASIC; in days of yore, (B-15)/2 would be correct. Since B-8 is a tad larger than B-15, just stick to B-8 and not worry about which specific software release quit replicating the file-header bytes on the tail end of BSAVE files.

For the sake of safety, a parting caveat is needed: An icon file about to be loaded must be able to fit into an already declared array; BLOAD does not concern itself with anybody else's fate. You can accidentally clobber your program if an incoming file overlays memory areas that it should not. Read Chapter 3 if you are interested in how arrays are stacked up in memory by the GW-BASIC interpreter.

For those motivated by such things as productivity and profit, this chapter has, hopefully, favorably enhanced at least one or the other. For those enjoying the graphics game, hopefully some of this has clarified the rule books, at least.

For pixel poking cowboys, the following program may be of use. It will draw a border around the screen, in SCREEN 1 or 2, no matter what WIDTH has been established. To be able to POKE pixels, rather than to use the high-level language capabilities of BASIC, it is necessary to know which bits to shoot at.

How to draw a bead on particular bit positions can be discerned from the arguments imbedded in the code. For the high priced adapters, they are valid for SCREEN 1 and 2, but not for modes 7 and above. If you are inclined toward the big game, change the DEF SEG to &HA00, and experiment with the offsets until you get what you want to see. And have fun.

```
 6000 DEFINT I
 6010 DEF SEG = &HB800           'CGA memory
 6020 SCREEN 1
 6030 FOR I = 0 TO 7999 STEP 80  'vertical lines
 6040  POKE I,240                'bits 4-7   cols 0-3
 6050  POKE I+8192,240           'bits 4-7   cols 0-3
 6060  POKE I+79,15              'bits 0-3   cols 76-79
 6070  POKE I+8192+79,15         'bits 0-3   cols 76-79
```

```
6080 NEXT
6090 FOR I = 0 TO 79              'horizontal lines
6100  POKE I,255                  'bits 0-7   line 0
6110  POKE I+8192,255             'bits 0-7   line 1
6120  POKE I+7920,255             'bits 0-7   line 98
6130  POKE I+7920+8192,255        'bits 0-7   line 99
6140 NEXT
```

Chapter 8 = FILES

Presumably you are using a version of the interpreter that has disk files capability. If not, what follows is not likely to be very interesting reading.

My assumption is, if you do not have a disk-based machine, you are probably not interested in reading anything in this book. The old cassette-based machines must all be in the silicon graveyards by now, and it is hard to believe that anyone makes much use of the cassette-only mode of operation of the newer machines that have the "cassette portion" of the interpreter burned-in, in ROM.

When is the last time any programmer had to use MOTOR? It is still in the language. We used to use it to start the capstan motor running, preparatory to processing data on magnetic tape. Perhaps we should be grateful that REWIND and the old "punched paper tape" commands finally passed on, to the big bit bucket in the sky. (MOTOR is foreign to QuickBASIC. It is still in GW-BASIC, probably because it was burned into the memory of so many ROMs just a few years ago.)

On the other hand, some old timers are sorely missed. Once upon a time, in the early days of disks, we could OPEN a drive as a device and access disk sectors on a relative basis. Which made it pretty easy to write tools to fix disks that had become corrupted. And to be able to control what got written where, and more importantly, when, exactly, data was actually output to the disk.

Hold on. This chapter is not a trip down memory lane. A few nostalgic remarks are useful, however, to set the tone for what follows. File processing in BASIC is tied to both, its history of evolution, and to anachronisms that defy any sense of logic.

One purpose of this chapter is to lower the bridge between DOS and BASIC. The language manuals read as if what can be done, and what is done, is unique to BASIC. Because most I/O done by BASIC are via requests to DOS services, programmers must know thoroughly, DOS data file concepts: No matter what language they are programming in.

Where files are, and when records are actually written in those files is managed by DOS. What is in files is up to you. Some

BASIC commands do fool with the data stream you read or send to a file; it is not necessary to use those commands at all. In fact, in DOS, nothing in any file identifies them specifically as belonging to BASIC or any other language. (There is no way to look inside a file and know, absolutely, how it came to be.)

A second purpose of this chapter is to describe how, in BASIC, to store, retrieve, and update data in disk files. How to do it quickly, and safely. Here the discussion is about general concepts. Canned coding techniques for implementing some of these ideas can be found in Chapter 13.

The language manual is organized as two books in one. One is called the "User's Guide", the other is the "User's Reference". Of these two, the "guide" is, undoubtedly, the most misleading piece of literature in the field of programming. Especially so, when it comes to doing data file processing.

For example: "Sequential files are easier to create than...." And, "... series of ASCII characters." Further: "Creating and accessing random access files requires more program steps... requires less room...stores them in a compressed format in the form of a string."

Here is my alternative introduction on the same subject: If you want to use variants of INPUT and PRINT to do file I/O, OPEN the file in INPUT or OUTPUT mode. If you want to GET and PUT, OPEN the file in RANDOM mode. Conversely, if you want to move around in a file, you must use GET and PUT. If you only need to process records serially, from the beginning of a file to its end, you may be able to read it with some form of INPUT, or write records with PRINT or WRITE. Maybe.

Whether one is easier than the other can be argued, but to no purpose. Which method to use depends on what it is you want to accomplish; to do a good job in either case is not especially easy. In fact, BASIC sometimes makes it more of a chore to do either, than it is in some other languages.

Whether the data in a file is "ASCII characters", or fields have "compressed strings" is up to you. And it certainly has nothing to do with BASIC. All bytes in all files are simply bytes. Whether they should be interpreted in groups (fields) as accounting figures, as text, or as "binary" (words) as in a COM or EXE file depends on the frame of mind of two people: He that created the a, and he that wishes to make use of it.

It is certainly possible, and practical, to OPEN any file on a disk in either mode. Often, both modes can be used on the same file at different points in time. Occasionally, even, both modes can be used to an advantage, on a single file at the same time.

Yes, using two different modes of OPEN on a given file is a little odd. Perhaps. It is also odd, what we have to do to get a machine to do what we want it to sometimes. Which is the root of my earlier remark that BASIC (and DOS) sometimes makes it a real chore to do data file processing.

Of these two, DOS is the nutcracker, more so than BASIC. With the exception of indexed-files -- which BASIC gives no help on -- most of what follows is true, no matter what programming language we use. (At the conceptual level at least--commands are spelled different in other languages, and they each have their own vagaries, but the fundamental problems are the same.)

From here on my presentation is organized along the same lines as data files are typically processed. In three different ways.

Sequential files: A file is read (or written) from beginning to end without "wandering around" or leaving any gaps.

Relative files: Data in a file are accessed anywhere within the file by use of a record pointer. In concept, this is no different than using a subscript to access elements in an array.

Indexed files: Access to data in a file is exactly the same as for relative files, but rather than having to use a number to find a record, an "index key" can be used. Thus, see that an indexed file is a relative file whose records can be found by referring first to an index, to get the relative position number of a record (stored with the key), then by using that pointer to access the desired record.

Notice that these three terms (commonly used by software types on the big machines) apply to how data in a file is accessed, not about what is in the bytes. There is also, usually, an implicit meaning associated with each of these terms about how data is organized in files: About the grouping of bytes into records and fields.

Notice also that the accepted meaning of these terms pre-dates DOS and BASIC. Why they insist on warping the English language is beyond me. What they call "random files" should be referred to as relative files. Little logical use can be made of data accessed on a haphazard basis (see Webster re: random). For a really good description of these three file concepts, read any COBOL manual.

Neither DOS nor BASIC has any way of knowing what is in a file. A file is simply a bunch of bytes. The terms used here are convenient for human dialogue; they infer how data is arranged in files, by a programmer.

Now is the time for all good coders to review which of these is correct for a BASIC program. To do so, we must know what BASIC does. Specifically. We begin with a look at sequential files. They are the simplest in concept, but most peculiar insofar as how BASIC treats them. Relative files are next; nearly all data processing in BASIC must be done with relative files. This is also fundamental to indexed files. Because BASIC does not have indexed file capabilities, per se, if we want to access records on the basis of keys, we have to generate and maintain our own "key index".

PS: Even in languages that do support indexed files, it is not uncommon for a programmer to do his own thing, to overcome real or perceived inefficiencies inherent to that language. This is probably more often done in the world of DOS than with other operating systems, regardless of the language used.

Sequential files contain ASCII character strings. So the books say. They are wrong. ASCII defines a range of codes from 0 through 127 (decimal). The other 128 numbers (128-255) that an 8-bit byte can represent are not defined by that standard, and are therefore, not properly called ASCII characters. These are codes for fancy characters that can be displayed on a monitor, and, printed by some printers. Bytes in the 128-255 range can also be read or written by BASIC programs to data files that are opened for INPUT or OUTPUT. Whether they should be seen as characters, or symbols, or whatever, depends on the eyes of the beholder and the legerdemain of the magician that stored them in a file in the first place.

The low-order ASCII codes, from 0 through 31, are designated as control codes, rather than as representing human-readable text

characters. In a narrow context, BASIC treats three of these as the standard intended, when reading or writing data in files opened in sequential processing mode. The specific codes that BASIC somewhat conforms to the standard on, are:

decimal	ASCII meaning	BASIC behavior
10	LF - line feed	end of field (weird)
13	CR - carriage return	end of field & record
26	SS - start special sequence	end of file sentinel

The other code possibilities in the 8-bit, 0-255 range are processed by BASIC no different than is the letter A (65), save for a comma (44), and a quotation mark (34), as we will soon see. My remark above that code 10 is weird is because

```
PRINT #1,CHR$(10);"hello";CHR$(10)
```

will output what we say, but if that data stream is read by

```
LINE INPUT #1,X$
```

then X$ will contain nine bytes. The values of those bytes in decimal are: 10, 104, 101, 108, 108, 111, 10, 13, and 10. What is peculiar is that code 10 is treated like any character, but it also causes the 13/10 pair at the end to be read into X$, as if they too were regular characters.

Normally, a 13/10 pair (0D/0A for those that like hexadecimal) work as an end-of-input-into-a-variable, and those codes never normally get passed up to a BASIC program using OPEN for INPUT. Which is also true of code-13, by itself, by the way, whether it is followed by a code-10, or not.

As a last act, PRINT to a file is always followed by the 13/10 pair. BASIC does this for you, whether you like it or not. Just like when printing, semicolons and commas between fields in a PRINT statement cause continuous strings of output; the semicolon and comma are merely syntax characters, they are not output to the file. (Unlike the semicolon, which does nothing, the comma causes extra spaces to be generated in the output stream.)

The WRITE command was invented a few years ago to simplify the problem of generating field separators in output streams. They are needed to be able to read sequential file data with INPUT

or LINE INPUT (with the file-number syntax) into separate variables used in a single statement. Like in:

```
LINE INPUT #1,X$,Y$,Z$
```

the incoming data stream must have commas imbedded in the data so that the data can be aligned with the separate names of your variables (fields).

WRITE will generate those commas on the same basis as they were used. Like:

```
WRITE #1,X$,Y$,Z$
```

will cause a comma to be inserted in the output file between each of the fields represented by X$, Y$, and Z$. Wow.

WRITE also puts free quotation marks before and after data output from string variables for us, so that on INPUT someday, the interpreter will not get confused by commas and quotation marks that are really text, i.e., that are not control codes that BASIC depends upon to align data bytes with variables. Wow, again. (It is hard to believe that anyone ever uses WRITE; none of my programs ever have.)

INPUT$ is another odd BASIC command, only sometimes handy for processing sequential file data. Like: X$ = INPUT$(1,1) which will read one byte--the next one in sequence since the last "read"--and assign that byte to X$, no matter what that byte contains. INPUT$ can be used to read varying numbers of bytes, in groups. It is necessary for the using statement to know how many to ask for, lest it ask for more than can be had.

The reference manual implies that INPUT$ is for communications processing. Beats me why. A file is a file, and bytes are bytes, no matter whether they travel courtesy of the telephone company, or are merely dizzy from riding around on a spinning disk.

In practice, OPEN for INPUT is nearly useless, no matter why you might be inclined to do it. No matter what is in a file, written by a BASIC program, or anyone else, coming from disk, down a wire, or right out of the ether.

The practicality of using OPEN for INPUT is severely limited by

what happens when we do INPUT or LINE INPUT or INPUT$ into a string variable. The cruncher is, any form of INPUT works like LET, which burns up string-space like we owned stock in the companies that make memory chips. (See Chapter 4. It is this little "gotcha" that, perhaps, motivated the observation that sequential files are slower than....)

Outgoing data can be shipped, after doing an OPEN for OUTPUT, with no hidden performance impacts. PRINT to a file is an easy method for generating variable length strings of bytes that are terminated by a 13/10 pair of bytes. If that is desirable.

That other byte generated by BASIC, whether we like it or not, is a code-26 (&H1A) when a CLOSE is done. It is the last byte written onto the tail end of a file that was opened for OUTPUT. It is what is called in the DOS manuals, a "Control-Z code". (It is amusing, but offensive, to see DOS manuals turn caret into a verb, as in: "... a careted character.")

This perquisite is an inheritance. In the early days of CP/M, which was the father that spawned DOS and its intermediates, a code-26 was needed as a sentinel to mark the end of a sequential input stream. The file directory mechanics in CP/M could not afford the space needed to know how long a file was, in terms of the number of bytes it had in it.

CP/M knew file lengths in terms of sector-bunches. (Which, incidentally, is the root cause of why the default record size in BASIC is also 128. Which was, also, the size of a sector in the earliest days of flexible disk technology.) DOS knows how many bytes a file has in it, no matter who wrote the file, in whatever language was in use, at the time the file was last written-to (and successfully closed by) DOS.

Because DOS knows exactly how many bytes are in a file, the old code-26 is no longer a demand of DOS, and therefore, it is no longer needed by BASIC, either. An end-of-file will be flagged when reading sequential files, by any program running around in the wonderful world of DOS, whether the file has a code-26 tied to its tail, or not. By definition, then, that archaic guard byte that BASIC still tacks onto sequential files when CLOSE is done is not needed, even by BASIC, when it reads its own files.

On the other hand, you get ejected from the DOS carrousel when a code-26 is spotted halfway around, or anywhere else, be it the end of a file or not, if you are reading serially from a

file that was opened in "ASCII mode" (DOS meaning).

Now, reread the DOS definition of its COPY command, and glean what the slash-A and slash-B options really mean. And, from whence their need has been perceived. Their use of the terms ASCII and binary are as ambiguous as they are in BASIC books. Of course, we need no reminder that DOS and GW-BASIC manuals were written with pens dipped in the same ink well.

Save for the BASIC-only peculiar fondling of commas and quotes, we can summarize nearly the whole subject of sequential files as being akin to what DOS calls ASCII files. Although they take liberal tradecraft license with our native language, the BASIC manual could simply say that if you understand the argot of DOS, OPEN for INPUT and for OUTPUT works just like DOS says, when it talks about using DOS commands such as COPY, and SORT, and EDLIN.

To me, with my penchant for the simple, nothing could be more straightforward than relative files. We can GET any byte we want to, and PUT one wherever we want, whenever the mood comes upon us, without worrying about which codes are likely to upset the apple cart.

Some would seem to revel in the sublime. Internally, BASIC has its own scheme for storing numeric values. (Chapter 5 pounds that pretty hard.) Because the technique that is used is very efficient for storing great big numbers in a minimum number of bytes, it is easy to understand why the manual brags that BASIC stores data in relative files "...in a compressed format."

Bragging is one thing, but they carry on with obfuscation about using MKD$, CVD, and their kin. Which need not have been said at all. Whether you opt to use those functions in conjunction with processing data stored in disk files is totally dependent upon what you perceive you ought to do. Which, in any event, has nothing to do with the fundamental concept of storing data in files that are to be processed on a relative basis. It is even possible, and practical, and a pretty good trick once in while, to PRINT #1,MKD$(A), for example. The choice, in any event, is yours; do not be misguided by a scribe that makes a living writing manuals, rather than by writing programs.

The quintessential aspect of relative files in BASIC is FIELD. When a GET is done, a string of bytes is copied from a DOS I/O

buffer into a BASIC buffer. The "standard way" to gain access to those bytes is via string variables that have been declared to be for that purpose. They are declared in FIELD statements.

The naming of variables in FIELD statements does five things:

1. The name-pointer is now aimed up into a file I/O buffer. (String variables normally point to literals in the body of your program, or, down into so called string-space; q.v. Chapter 4.)

2. The structure of what is expected to be in records, in this file, is described (to yourself) by the order in which variables are named, and the field size stated with each name.

3. Each named string variable is explicitly defined as a fixed-length string of bytes. They enjoy a substring posture (of the total string that represents a record), relative to the order in which they are stated in FIELD.

4. In concept, GET is like using LET several times: Each variable named in a FIELD statement will contain an image of what was read from a disk, as a substring of that contiguous string of bytes then in the file I/O buffer.

5. When a PUT is done, the entire buffer is passed back to DOS, so that it can overwrite the area of the disk from whence this "record string" came from. While data is in your buffer, whichever fields (variables named in FIELD) you opt to fool with, will be "updated" on the disk. See that the content of variables that you do not change go back to the disk, just as they were.

My attempt to describe what you probably already know is not as succinct as could be done by a pro author, but it is much more correct than the opening verses about files in the BASIC hymnal called the "User's Guide".

Whether you prefer the argot of the manuals or my awkward stab at using conventional English is unimportant. Communication is, however. There are a number of things that are not said at all in the manuals. The whole truth needs to be known if they actually expect us to make intelligent use of what is, actually, a powerful language product.

It is unlikely that a gun manufacturer would be guilty of not describing safety features of his product. Misuse of BASIC is not likely to cause you to accidentally shoot anybody. There is a real risk, however, that a badly written file processing program could cause an operator to want to strangle you.

In the interest of filling in the voids, here are some more notes compiled from the scratches penned in the margins of the pages in my manuals.

OPEN: Do not use the archaic form of syntax, the one where you use letters like "I" and "O" and "A" to declare a file access mode. It does not behave exactly as does the modern form of syntax, despite the manual's promise. Re: You can OPEN and CLOSE a file (number) as many times as you want to. No you cannot. If the old fashioned form of OPEN is used, an error may be provoked on the 16th attempt to execute OPEN. This bug is wobbly, and crops up only at certain times, but it does not really matter when. There is no good reason to dredge up dying bugs.

LEN: State it explicitly, even if 128 is the desired number. An error trap will occur if this number is larger than that specified with a slash-S when GW-BASIC is loaded. If not stated, an error is not flagged until a FIELD statement is encountered that attempts to exceed 128, or, the global size maximum specified when BASIC is jump started. It is better to have an OPEN fail, than to crash later, leaving possibly, a fractured FAT on your favorite disk. (PS: QuickBASIC has no counterpart to the slash-S option.)

LEN: The scribe that said you could specify 1-32767 never tried it. If you try to start GW-BASIC with /S:32767 it will not work. Even if you also specify /F:1, you will still get a DOS message: "Out of memory". Even if you have a couple of million bytes of RAM sitting empty. The practical limit for LEN is about 8kb, which will leave you enough room in BASIC's 64kb program-bucket to run a few lines of code. Bear in mind well, whatever slash-S option is specified, it will apply to all files once the interpreter is loaded.

FIELD: A variable-name may be used more than once in the same statement. Example: FIELD 1,3 AS X$,4 AS W$,2 AS X$. In this case, X$ is 2 bytes, beginning at position eight, counting from the left. The first X$ is similar to FILLER in COBOL. The reason for doing this in BASIC is to obtain the

desired alignment of name-pointers into a buffer, without concocting artificial names that are not going to be used. (Chapter 3 explains why we should coin no more variables than are actually needed.)

FIELD: String-array elements can be named just like discrete variables. In fact, this is great for updating records with FOR/NEXT loops. There are a few pesky points to ponder, however, when opting to go this route:

+ If an array subscript named in FIELD is larger than 10, a DIM must be done before this statement is attempted.

+ It is the individual elements of an array that are named in a FIELD statement that are associated with a file buffer, not the entire array. Looping logic that whacks string arrays must take care to not abuse subscript positions sacred to a file. (It is not altogether a bad technique to have one array do double duty, where, some spots are "normal" strings, and some equate to fields in data records.)

+ In the magic kingdom of the compiler, all fielded names, and the data they point to, are abandoned on CLOSE. This has a side affect, in the case of string arrays, of being similar to an ERASE, but of only those array positions specifically named in a FIELD statement.

+ ERASE and DIM work as would be expected. If FIELD uses array names and discrete variables, also, the pointers into the buffer for the simple variables maintain their relative position; the substrings previously pointed to by array declarations are "logical gaps". ERASE does not "erase" the data contained in the I/O buffer. Naturally, DIM done again for the same array means that all of its subscripts now point into string-space, notwithstanding whether some may have previously been aimed into a file buffer.

LOC: If you GET 1,5 (or PUT 1,5) then LOC(1) will report 5 as the current record pointer. Just like the books say. If you GET 1,32767 the LOC(1) report is correct, but GET 1,32768 will cause LOC(1) to report -32768; and you cannot PUT to a negative record position. If you GET 1,65536 then LOC(1) comes up zero, and you cannot PUT to zero, either. Worse still, GET 1,65537 followed by PUT 1,LOC(1) will overwrite record number 1, not record number 65537. Annotate your

manual in red: LOC is an integer inside the interpreter. It is automatically reset to 0 each time you pass 65536, and, reports negative complements of 65536 for records in the range 32768 through 65535. Meaning: LOC is pragmatically useless (dangerous) for a lot of files.

LOC: In QuickBASIC the above paragraph does not apply. If you go to the extremes necessary to solve the LOC problem for an interpreted program, all that work is for naught if you later use the compiler. The simplest solution is to adopt a habit of maintaining your own GET/PUT counter; use either a single or double precision variable, depending on the growth anticipated for a given file, and not worry about LOC limits.

LOF: The length of a file is always reported as the number of bytes in that file. To equate this to how many records are in a file, divide LOF by the LEN argument used in OPEN. That advice in the manuals is valid and accurate. Make it a habit to use the forward-slash in the division expression. The back-slash is used in BASIC for integer division; the compiler will not tolerate this for an expression involving LOF. (The compiler thinks it is smarter than you. You know the answer is going to be small enough to fit into an integer variable, but the compiler designers assume that mere peons cannot possibly know something that they do not know.)

GET: Works just as expected, just as advertised. It is not intuitively obvious what happens when you GET farther than you have ever PUT, however. In a new file for example, if the first PUT is to record number 10, what is in records 1 through 9 may be anything; even, leftovers from yesterday's lunch. Logically, records that you have not explicitly PUT with known values have to be treated as if they contain garbage.

PUT: See GET, above, then remember that PUT merely sends a record to DOS; it DOES NOT mean that it will go to disk, now, eventually, or ever. (If you OPEN a file by two different numbers concurrently, do not assume that GET on one will look like what was PUT to the same file known by a different file number. Caveat, Jose.)

CLOSE: Believe it or not, you can CLOSE files that have never been opened. Funnier still, you can CLOSE file numbers that can never be opened. (CLOSE is tolerated with any value of from 0 to 255. Anytime.) The manual says that CLOSE also writes the final buffer of output for sequential files. What

it does not say is that there is no way of knowing how many relative file records have not yet been written to disk. You may have logically updated dozens of records. The fact that you closed a file means nothing. Updated records may still be floating around in DOS buffers; DOS sends them to disk whenever it takes a notion unless it is told to do it right now. And the only way to get BASIC to tell DOS to unload its cargo immediately is to issue a RESET command. Unfortunately, you cannot RESET files by number. Which means you may have to do another OPEN, even for read-only files that have not been changed.

RESET: The reference manual says to always do this before removing a diskette: "Otherwise when the diskette is used again it will not have the current directory information written on the directory track." Small advice from small minds, for small disks. In the event of a power failure, for example, even if your program had already done a CLOSE on all files, the FAT on a hard disk (which you cannot remove) is apt to have been clobbered. (The File Allocation Table is what CHKDSK looks at. When it reports broken chains, it is not unusual to find that a BASIC program broke them.)

RESET: Issue CLOSE, then RESET. As a habit. This will save you a lot of grief, on those days you opt provide an operator the option of sending data to a printer or a disk (by the simple expedient of using "LPT1:", or a file name, in an OPEN statement).

My notes above stem from things found out the hard way. This list should be longer. It is very time consuming, and not very productive, to have to reconfirm or refute this knowledge by trial and error, every time we are forced to upgrade to a new release of software.

It is also possible that you know of many things that could have been illuminated here. My knowledge is derived from the way my programs work, and from the experience gained when some did not work as anticipated. And that experience makes me gun shy. A number of things have been added to BASIC in recent years that I make no use of and therefore do not know, or care, how accurately they are described.

QuickBASIC changes are so anachronistic no mere prophet can anticipate what it is apt to do next. Once you have mastered most of its nuances, stay with that release as long as you can.

Unlike GW-BASIC, you have to keep "old" compilers around the house forever, to be able to recompile old programs. Either that, or suffer the burden of having to make a lot of otherwise unnecessary changes, just because of some whimsical change the compiler writers thought was a good idea.

Although changes to GW-BASIC are somewhat infrequent, because it, and QuickBASIC, are so closely tied to DOS, it is DOS we have to try to keep up with. And it changes rather frequently. When they added path commands to BASIC, like CHDIR, it was not intuitively obvious that the potential advantages for using it would offset maintenance headaches if some future change in DOS invalidated an application program's internal logic.

> Note: The upgrade force-factor cannot be avoided altogether. Clients buying new computers cannot buy outdated releases of operating software, even when older versions would be fully adequate to their needs. We cannot simply give them a copy of our old versions without violating copyright restrictions. Catch-two-two.

It is the sum of hard-won lessons learned that causes me to suggest the following "attitude" to adopt about doing file processing in BASIC: Do what needs to be done with the content of files, inside your program. Do disk maintenance, general file mapping, and systems configuration tasks externally, with DOS batch files, or whatever. And refuse to have anything to do with "networking", in any language, unless you have a key to Fort Knox, or have a client that does. File sharing ideas help sell iron; it is a lousy concept for responsible systems analysts to echo. Especially on DOS-based machines.

Even if you buy this advice at face value, you can still get badly burned. Here is a very recent history lesson that shows why it is better to be conservative, rather than risque.

> VDISK (alias RAMDRIVE): The initial state of all records in files on a symbolic disk (in memory) is all zero-bytes. How long we can count on that, is unknown. It might be best to not have blind faith. Right out of the clear blue, DOS 3.3 changed the name of VDISK to RAMDRIVE. We have not fully recovered from that nasty trick, yet.
>
> There is no way of knowing for sure how many CONFIG.SYS files are going to flip-out when an unsuspecting user decides to

"update" his DOS. A program you wrote may run for hours before you have need to hit the soft disk. Then you crash. It never even crossed your mind that DOS would take away something that has been there for a number of years.

Hint: Now my CONFIG.SYS files attempt to load both VDISK and RAMDRIVE, betting that one or the other will be there, and, betting that DOS will continue to merely display an error message and proceed blithely on, fast enough, that nobody will notice the file-not-found message during boot-up. (Or that one of them will flash an "Incorrect version" message because the user simply file-copied new DOS files over the top of the old ones, causing both to now be resident because their names are different.)

Enough of the grit, gripes, gloom, and doom. It is time for some real fun: Indexed files. This is where your creative instincts can really come to the fore. Especially if you want to obtain maximum performance with the least effort and risk.

The first requirement to ponder is whether an indexed file ever needs to be processed sequentially, in sorted order, based on their keys. Most do, sooner or later.

ISAM--Indexed Sequential Access Method--is a concept as old as computers. This acronym has been in my vocabulary so long I cannot remember from whence it originated, but the meaning has always been essentially thus:

1. Keys are maintained in an ordered sequence. Often, for example, in ascending alphabetical order.

2. A find-attempt--such as when an operator enters a customer account-code--hands up that target record if an exact match is found for the requested key.

3. If a requested key is not found, the record for the next higher key in sequence, is targeted. (Whether or not it should be shown to an operator depends on why they are wandering around in a file in the first place.)

4. Sequential reading of a file is possible (from any point in an index) by simply stepping forward through the index. Thus: Customer lists can be output in sorted order, for example, with no need to do a preparatory sort job.

A byproduct benefit of ISAM can be that it makes it easy, and quick, to preclude duplicate keys. Occasionally an application's needs dictates that duplicate keys must be allowed. That can be, of course, but it is a real bear to program for. In the absence of specific demands to the contrary, everybody will be happier if you design so as to prohibit duplicates, i.e., all records have a unique key in the index.

To this point, ISAM is a classical scheme, in any language, on any computer. In fact, add-on packages can be purchased that can be "called" from BASIC, to do all of this, and more, and they do all of the housekeeping necessary to maintain even multiple indexes, and alternate-key accessing mechanisms.

In many cases such add-ons (purchased or homemade) are as inefficient as those that are built-in in other languages or operating systems. Because of "parameterability". They spend a lot of execution time doing the equivalent of IF. And they spend a lot of time running around the disk looking for your records.

By reducing an application's requirements to specific needs, and by coding explicitly to that end, even interpreted BASIC programs can perform more efficiently than some "generalized" software, no matter what language it is written in.

This does not mean we have to reinvent the wheel every time we write a new program. By having a library of subroutines for different "functions" associated with file I/O, they can easily be merged into new programs on an as-needed basis, and quickly parameterized, specifically, in-line, in the code itself.

Here is a list of functions needed to manage indexed files by my definition.

Find: This subroutine searches the keys-index for a file. On return, it points to an exactly matching key-entry, or, to the next one down, in sequence. If the requested key is "larger" than the last one in the index, the pointer is set to the end of the index, plus one.

Add: This subroutine inserts a new key into an index, at its proper position, according to the logic that governs how keys are supposed to be ordered. And it shifts all of the keys that follow this one, down by one position.

Delete: This subroutine deletes a key from an index, and moves all of those that follow it, up by one position. It may also move the one deleted to the bottom of the index and mark it "deleted". The pointer tied to this key can be an easy way to be able to reuse obsolete records, to curtail unnecessary growth of highly volatile files.

Move: This subroutine calls Delete, then Add. This is done to resequence keys that are being changed, i.e., the record is being kept on file, but the spelling of its key is being changed.

The key to managing keys with this functional approach is the Find subroutine. It can be called to merely locate a record, and it can be called by Add, Delete, and Move, as needed, to find relative positions in the index itself, for doing index maintenance.

Typically, Find uses some variation of a binary-search. The exact technique to use has to be based on how keys are ordered in an index, of course. Many excellent books can be found that describe various searching and data ordering schemes. Few can be found that address problems specific to doing tasks like this in interpreted BASIC, in DOS, however. Chapter 13 contains some specific solutions for this.

Two things that deserve special attention in our world is the overhead burden suffered by BASIC programs on each reference to a variable, and, the double-buffering of records in memory that is done by GW-BASIC running under DOS. Chapter 3 describes the variables-search problem. The other major concern, about how records are held in memory, once they have been read from disk, is appropriate to this chapter. It is apropos to all relative file processing, but emphatically so, when designing routines for managing indexed files.

Working from the bottom up, so to speak, DOS disk mapping and record buffering, then how records are buffered by BASIC, then how the interpreter works internally, then an application's requirements have to be weighed, before a method can be chosen to get the job done efficiently.

On a GET, a lot of bytes get moved from one location in memory to another, even if an OPEN specified LEN = 1. Like this:

1. Application program tells BASIC to GET one record.

2. BASIC tells DOS to GET one sector.

3. DOS tells the adapter which sector to read, and where in memory to put a copy of that sector.

4. DOS tells BASIC where in memory that sector's image is.

5. BASIC copies that portion of the sector that represents your record into its file buffer.

6. Your fielded variables now point to fields within your record, in the interpreter's file buffer.

On a PUT, all of this is reversed. The interpreter copies your record back into its corresponding position in the DOS buffer, and tells DOS to send that sector back to disk. (DOS does it when it takes a notion, remember, which may only occasionally coincide with when you said PUT.)

My newest DOS manual says: "Feel free to experiment with different buffer settings...." This innocuous quotation is a NOTE on the page describing BUFFERS, following another that says, a number between 10 and 20 provides the best performance for word processors.

Certainly we can feel free to experiment. We own the machine, and we paid for DOS. Reminding us of our consumer's rights provides little help. Telling us to experiment is ridiculous, also. What we need to know is how DOS makes use of all of those 512-byte buffers, whatever number we opt to try. (Our memory cost is 528 bytes, per buffer, notwithstanding the 512 stated in the books; 512 is for the data, and 16 more bytes are used, per buffer, for keeping track of which file each buffer belongs to, and its usage indicators.)

The underlying queuing theory of DOS record buffering seems to be FIFO: First In, First Out. None of my manuals ever describe any of this, however. Probably because it is not apt to be the same for any two versions of the software. (Viz, the onerous trick they pulled in Release 3.3 when they changed the number of default BUFFERS. Programs that used to work may no longer, because all of a sudden you are out of memory; at least until you recode CONFIG.SYS to specify the "old" BUFFER default.)

Here is my perception of how FIFO buffering works in DOS, on this Friday afternoon, anyway.

+ On any program's request to READ, an image of the sector to be read will be placed in the next available buffer that is thus far unused.

+ After all buffers have been filled, from then on the "next" buffer to be used is the one with the "oldest" contents.

+ In the event something has been changed in an "old" buffer, it will be written back to disk, from whence it came, just before that space is overwritten by the latest request to fetch a different sector.

+ In the event a READ request is for a sector that is already in a buffer, no disk I/O is performed, but this request is treated as the "most recent" by the little gremlin that is keeping track of which is the "oldest" inactive buffer.

Add to this jungle savvy: DOS uses these buffers too. Like for reading disk directory information and FAT fiddling. And where some wag came up with the idea that word processors run better with lots of buffers is inexplicable. (Of my three WP programs, none seem to work better with two buffers, fifty, or any of the numbers in between.) My older DOS manuals are even more obtuse. They predict "data base" applications work better with lots of buffers. (I hope that writer get "lots of" pay.)

What all of this really boils down to is, sequential reading or writing of large chunks of disk data benefits not a whit from multitudes of buffers. Nor does GET and PUT activity involving but a single file. Nor does GET and PUT for many of the files that may be open concurrently, if one is constantly hogging the show by being hyperactive. Which is a typical characteristic of those known as indexed files.

Now we are ready for another unadvertised piece of lore for the folks using BASIC. Double buffering. This has been alluded to already, but see it now for its performance implications. With pieces of disk-data sitting in DOS buffers, and chunks of that sitting in your GW-BASIC program's own buffer, and maybe some of that parceled out to different variables for interim uses, it is going to take some time, sometimes, to put it all back where it belongs. Which brings us to the point of thinking about the program we are about to write.

The first thing to decide upon when analyzing an application's technical requirements is: Record sizes. The very worst thing we can do is pick an odd number (other than one, maybe).

Consider the implications of a record size of 513 bytes. The odd byte, the one after 512, means that for every GET, two sectors will have to be read from disk, which will use two DOS buffers. And every PUT will have to unload two buffers, if the field you "updated" crossed the magic divide. And see that 255 is just as bad as 513. Or maybe worse. Yet, in GW-BASIC, the maximum length permitted for a string variable is 255. (This restriction applies equally to variables named with FIELD. The sum of the sizes of the several fields described with a single FIELD can be greater than 255, of course--up to the total specified with LEN, on OPEN.)

So, design rule number one has to be: Record sizes should be a number evenly divisible into 512. Preferably, no record should be larger than 512, either, if that file is going to be open continuously, and experience GET and PUT activity on a sporadic basis.

Not all record-size requirements are easily forced to conform to this advice. So, round upward to the next higher increment that will adhere to this rule. Even, if you wind up with a few bytes tagged as "reserved", just like the big boys do. They may come in handy, anyway, when your client suddenly remembers something he forgot to tell you about his needs, originally.

The other choice for forcing record lengths that will be some figure that will preclude spanning across sector boundaries is to opt for "compacting" one or more data fields. Here, the trade-off has to consider the processing time required for packing and unpacking such data, and the frequency with which it must be done.

The next decision to be made, for indexed files, is the one likely to have the greatest impact of all: Where to keep the index itself. One of the obvious possibilities, of course, is inside the related data file. At the top, or at the bottom, or as "pages" interspersed among blocks of real records.

Many indexed file schemes, in a lot of systems and languages, keep the index for a file at the front of that same file. It is obvious, also, why most such schemes dictate uniform key

lengths, and why, the maximum length of a file (plus its index) must be specified at the time a file is first created.

An alternative that easily permits variable length keys, and unlimited file growth is to keep the index in one file, and the data records in another.

Any scheme other than one that prewrites an arbitrary block of sectors at file-creation time has to anticipate performance degradation will be experienced sooner or later, because of that old DOS Nemesis known as fragmentation.

One of the best methods that can be used to achieve superior performance for indexed files is MRI: Memory Resident Index. This is another old-timer's term, dating from so far back its origin escapes me. It was good technique back when, and it still is today.

On start-up, load a file's index into memory, then close that file and open the real data file. At the end of a run, if the index has been changed, dump the updated index back to the disk.

A chief advantage to this scheme is that no disk thrashing will occur while you are wandering around in the index; all I/O done during a live performance is done for the benefit of paying patrons. A disadvantage to this is, the auditorium must be large enough to hold the entire index, backstage. Still, this trick should not be discarded too quickly. See this:

```
DEFDBL A:DEFINT I                'define data types
DIM A(2000),I(2000)              'A = keys; I = pointers
BLOAD "keys.ndx",VARPTR(A(0))    'load the keys
BLOAD "ptrs.ndx",VARPTR(I(0))    'load their record pointers
```

Total memory needed for a 2000-record data file is just a tad over 20,000 bytes; 2000 times 8-bytes for the keys, and, 2000 times 2-bytes for the LOC-pointers into the real file. Yes, this works for "alpha keys", also, of up to 8-bytes each, by using MKD$ and CVD on each of the slots in the double-precision A-array.

Using BLOAD to grab the index, and BSAVE to unload it is very fast. Even an impatient operator will not know when you did it, especially if it is being kept on a hard disk.

Another crafty alternative is to COPY a file's index from disk, to VDISK, before a run begins, and to COPY it back to disk at the end of the run, if the index has been modified. Which is one more argument in favor of keeping a file's index in a file separate from the one that has the real records in it.

Last, but not least, do not discount too quickly the easiest scheme of all: A no-index, indexed file. It is so simple to do, it needs no acronym by which to remember it as a concept.

Simply maintain the records in a file in sorted-order, and use a binary search, or whatever, to find a desired record. My previously described functional approach is valid here, too, but see that only the one, real data file is involved.

In the final analysis, files with upwards of two or three thousand records can be "sorted" fast enough today, to not irritate the operator in a lot of applications. Especially at the speeds we are enjoying now, on "modern micros". Limits like 3000, for files that experience infrequent additions and deletions, are forever being extended by faster clock times, increased data-bus bandwidths, and speedier disk drives.

In fact, it is past time for DOS to consider dispensing with the whole crazy business of BUFFERS. Once upon a time it was a software bridge for overcoming slow mechanical devices. Just as "virtual memory" was once a clever, but highly complicated substitute for high-priced core memory.

How sweet it will be, when the disk finally spins full circle, and PUT will do what it did on the first generation of micros: Update our disk data, right now, at our command.

Meanwhile, we have to continue to worry about what happens if a failure occurs. If we update an index first, and a failure occurs before DOS gets around to updating the real record, the logical consequence is crap-O. If we update the real data file before we update the index, the result may still be a big mess. Worse still, if we succeed in doing both, but a failure occurs before DOS fiddles its FAT, everything on the disk may be one big-a-bunch-a-crap-ola. (Again, Chapter 13 gives several CYA solutions for this problem.)

My wonderful wife of thirty years, our five grown-up offspring,

three lovely daughters-in-law, and three precious granddaughters all have faith that this old man has figured all of this out correctly. When they close my file, hopefully my sins will be remarked as having been no worse than, an irreverent view of the software that fed a lot of us, a lot of years. My legacy to my peers is in these pages. May you avoid some of the pratfalls I have taken, processing data files on DOS disks, in BASIC.

Chapter 9 = STRANGE

It is a fact that, if that is what the computer says, it must be so. Ask anybody. Everybody knows how accurate computers are. Similarly, if the manual says, if you do such-and-such, the machine will do so-and-so.

Both of us know better. Our ability to write programs (in any language) depends on our understanding of what the manuals say, and, the extent to which our experience says those books are right, almost right, ambiguous, or in fact, wrong.

Both of us know there are bugs. In the software we write, and in the software we buy. And in documentation, theirs and ours. Known bugs we can all learn to live with. Unknown gremlins, however -- in documentation or software -- can cause more head scratching than a colony of fleas on a mutt. And make us just as sore. And cost us time and money.

My outline for this book presumed that various "facts" could be presented at appropriate points in each of the chapters dealing with specific subject areas. As the meat hit the grinder it soon became obvious that not all that has been learned could be dished out that way. Some phenomena defy all attempts to offer a structured presentation. This catchall chapter was squeezed in, to cover the gaps.

Elsewhere my remarks reflect my cynicism about a misnomer or inadequacy in the manuals. Here, the truth (as I perceive it, as a research staff of one) is meant to be helpful when you too experience strange encounters: When, what you see is not what is supposed to be.

In some cases the mystery is because of a bug. Perhaps. Many times it is simply because that is the way it works. Whether that is the way it was supposed to work, or not, is not really important. What we need is to be able to solve such mysteries only to the extent that we can continue to program with a sense of confidence in ourselves: That we know what we are doing and, what our program will do in the event of.... The notes that follow list some strange encounters in my time on this planet.

In any programming language, we would expect that movement of

data from one variable to another would produce an exact copy in the target of what was in the source. By inference then, when the target and the source are one and the same, what we would expect is, effectively, no change. Thus A = B causes A to have the same content as B. And, A = A should cause A to not be changed, at all. BASIC is consistent with this age-old principle of languages, with one remarkable exception:

```
1000 DEFSTR M-Z
1010  X="0123456789"
1020  Y="0123456789"
1030 MID$(X,3)=X      'repeats 1st 3 char for LEN(string)
1040  PRINT X
1050 MID$(X,3)=Y      'overlay 1st 3 char of another string
1060  PRINT X
RUN
0101010101
0101234567
```

Notice that line 1030 above names X as both the target and the source. In line 1050 X is the target, Y is the source. For a given expression, we would expect consistent results, no matter what variables are named.

Now we can illuminate our manual with this peculiar trait of MID$ and keep on trucking, with one of two attitudes: Be alert while debugging; what we thought would work may not. It is also possible that this "undocumented feature" can be used now and then to our advantage. If, for example, a string is needed that contains a repeated sequence of characters, we can use a trick modeled after line 1030 above to generate a longer one automatically, using merely the few characters we want to have replicated.

Speaking of attitudes: There is an attitude reflected in most software manuals that says, effectively, if you opt to take advantage of any "undocumented feature", you do so at your own risk. What they mean is: Bugs exist in the software and the manuals. Someday we may fix them.

This particular bug (my definition) is an omission in the BASIC manuals. The software has worked this way since the "upgrade" from MBASIC to GW-BASIC. It is also the way QuickBASIC works. It is unlikely they would change the way the language itself works now because it has been this way for so many years. It would be nice if they would at least update the manuals to tell

it like it is, however, and document this undocumented feature.

Here is another anomaly. It ought to be fixed. It is common to both GW-BASIC and QuickBASIC. No good use can be made of this bug, but, it can cause some strange encounters if you are not aware that it exists.

```
SOUND 0         'is supposed to be illegal, but is ignored

SOUND .49,99    'is supposed to be illegal, but clicks

SOUND .5,99     'is equally illegal:  it will cause an ERR = 5
```

By definition, the first argument used with SOUND can range from 37 through 32767. So the manual says. My reading of that says, anything less than 37 should cause an "Illegal function" error.

If my understanding of law is similar to Perry Mason's, it can be argued that nowhere was it promised that if we do something illegal, we will get caught by an error trap. Having read the manuals from front to back many times over the years, however, my inference is: We are supposed to be able to depend on the language processor to trap any errors we make that violate the rules of the language. Included in this assumption are things like getting an ERR = 5 if we use any argument anywhere that is not within the range permitted for a given function or statement. Oh well.

Philosophically, an interpreter or a compiler should be adept at catching syntax errors. In BASIC they both do, although they are not equally consistent in determining what constitutes errors in spelling or grammar. This is not hard to live with. Sooner or later, presumably, we can find all of our errors of that type, with or without their help.

We should code in such a way as to preclude trying to make use of "data" that exceeds the ranges permitted for various types of expressions. No argument. We are encouraged to depend on the language processor to "trap" our run-time errors, however. Numerous examples are given in the manuals that do this. It is their "recommended way" to detect when we have reached the end of a file, for example.

And that is bad advice. My advice is to do everything that you

can to avoid being caught in an error trap for any reason. To ignore this advice is bound to bring on some strange encounters sooner or later, if you want to do a RESUME NEXT. It is an "undocumented feature" of GW-BASIC that, it is hard to predict just where NEXT is.

The manual says RESUME NEXT means the program will continue at the first statement following the one that caused a branch to your ON ERROR GOTO address. That prediction is wrong, as shown in the following examples.

```
3030 ON ERROR GOTO 3040 : GOTO 3050
3040 C = E : E = ERR : RESUME NEXT

3050 IF 1=1 THEN ERROR 101:PRINT 1
3060 IF 1>1 THEN PRINT 2 ELSE ERROR 103:PRINT 3
3070 PRINT E    'output = 1 103
```

Here, ERROR 101 in line 3050 did resume at the next statement after the error, as would be expected. If RESUME NEXT worked as the manual says, however, we should also see a 3 because of the PRINT following the ERROR 103 in line 3060. In this case, RESUME NEXT after an error following an ELSE means go to the next line, not the next statement.

This one is even harder to fathom:

```
3100 IF 1>1 THEN PRINT 4:PRINT 2 ELSE ERROR 103:PRINT 3
3110 PRINT E
```

Output: 2 103. In this case NEXT went to the last statement to the left of the ELSE in line 3100, although the error itself was the first statement to the right of ELSE. There is only a small difference between lines 3100 and 3060. One has but one statement preceding THEN, the other has two.

Now the plot thickens. Using the same error handler above:

```
IF 1>1 THEN PRINT 4:ERROR 102:PRINT 5 ELSE ERROR 103:PRINT 3
```

followed by PRINT C;E will print 5 103 102, indicating that the ERROR 103 caused RESUME NEXT to hit PRINT 5, by backing up one statement to the left of ELSE. Because there was also a static error inside the THEN-clause NEXT effectively backed up one more statement and trapped that error. NEXT after that one came back to the next line (otherwise a second 5 would have printed because that is what is next, following the most recent error).

Even Dame Christie's Hercule Poirot would have trouble with that little mystery. And this next one, as well:

```
IF 1>1 THEN PRINT 0:X:PRINT 4:PRINT 5 ELSE ERROR 103:PRINT 3
```

The "X" after PRINT 0 is an obvious syntax error. This line will cause 4 and 5 to print, notwithstanding that the truth of the IF expression should skip everything after THEN, and run straight into the error that comes after ELSE. But it never gets that far in this case.

The next two examples are enough to make the point: Compound and complex conditional expressions merely serve to compound the mystery of just where NEXT is, when errors are imbedded in lines such as these. For the sake of easier eyeball tracking, these two lines are stacked as they might be in QuickBASIC:

```
3190 IF 1>1 THEN PRINT 2
            ELSE IF 1>1 THEN PRINT 2
            ELSE ERROR 103:PRINT 3

3200 IF 1>1 THEN ERROR 7
            ELSE IF 1=1 THEN ERROR 103:STOP
            ELSE ERROR 8:PRINT 3
```

Both of these lines end up printing 103 when we say PRINT E. That code was saved in E by the E = ERR in the error handler. RESUME NEXT goes to the next line, not the NEXT statement. The STOP in line 3200 is never even seen, although the error does immediately precede an emphatic command to go no farther.

Here is my attempt to condense the above empirical examples into a meaningful definition of RESUME NEXT:

For any error in a simple line, NEXT does mean the next thing following that error. Errors in lines that have conditional expressions disrupts the interpreter's lexical parsing of THEN and ELSE clauses. Errors after an ELSE will RESUME NEXT to the next line (not the next statement), unless there are multiple statements between THEN and ELSE. If so, RESUME NEXT is to the first statement preceding ELSE. All of which presumes there are no "static" errors preceding THEN or ELSE.

The distinction about static errors is important. Because the interpreter scans left-to-right, on a given line, an error may

or may not be seen following an IF. If the truth of an IF indicates everything after THEN should be bypassed, it will be skipped if the interpreter's byte-pointer does not run afoul of what is logically expected following a given token. (Chapter 2 describes the mechanics of parsing a program line in memory.)

On the other hand, "Illegal function" or other types of dynamic errors that could be provoked inside a THEN-clause that is not being executed will remain unnoticed. The upshot of all of this is, NEXT, for RESUME NEXT following errors that occur in lines that contain THEN and ELSE is sensitive to the type of error involved, the construction of conditional clauses, and whether more than one error is present, or potential, on that same line.

A literal translation of this evidence could be: Do not use IF-THEN-ELSE. Or, if you do, make no mistakes. In real life, the pragmatic rule should be: Code no I/O statements after an ELSE. If done after THEN, do no ELSE, or another IF-THEN on that same line.

This pragmatic philosophy is based on the fact that we have to depend on an error handler to trap I/O errors. There is no way to find out if an OPEN, GET, or PUT or similar things are successful until they are attempted. If after the fact, one of these does indeed fail, we must react accordingly.

The logical choice in most cases is to pass an I/O error-code back to the routine in charge via a RESUME NEXT. Doing so, it is imperative that the IF that tests for a failure has a chance to respond. By always coding it on the line that follows an I/O attempt, and by keeping the I/O command-line itself as simple as possible, we will have a pretty good idea of just where NEXT is, on a RESUME NEXT.

Not being aware of how RESUME NEXT does in fact work, you are in fact likely to experience some strange, strange encounters.

Now none of this confusion exists in QuickBASIC. It does as the book says: RESUME NEXT is always to the next statement following an error-trigger, no matter where it is encountered. But their skirts are muddy in another way: Not everything that can be error-trapped in GW-BASIC is possible in QuickBASIC. At the same time, ON ERROR in that language will trap some that GW-BASIC chooses to ignore.

There are two reasons for this, but none of the BASIC manuals

openly admit it. The two languages are not identical in terms of the semantics of the language. The permitted range of the arguments that can be used for many things are different; that can be seen, true enough. Although it is not conspicuous, we can deduce how this can affect logically coping with ERR = 5 errors. The other reason for some differences is that calls to DOS are not handled in exactly the same way in all cases.

A simple example of a difference that should produce the same results was mentioned in Chapter 6: WIDTH 40 is illegal on any machine with a monochrome adapter. GW-BASIC will do ON ERROR and give an ERR = 5. QuickBASIC simply ignores this type of error, and your program keeps right on humming as if nothing untoward had been attempted.

Even when we are alert to the potential for strange encounters caused by differences in the way these two language products interface to the outside world, we must know more. The manual for the compiler touches lightly on the differences about how ERL testing works, and how ON ERROR GOTO is supposed to work.

None of the manuals tell us that there are differences in what will trap, and what will not, however. In fact, they all say simply that ON ERROR GOTO works for any error that can be detected; never do they give us a list of what the detectives actually look for. All of which brings to mind stories about folks like J. Edgar Hoover and the FBI.

Strange encounters and unsolved mysteries are stories of one kind. This next one is in a class all by itself: Weird. It will put to rest that myth about "Seeing is believing...."

```
1000 PRINT "hello"     'a
1010 PRINT "and"       'tiny
1020 PRINT "bye"       'program
RUN
hello
and
Ok
```

Before we explore why this tiny example did not print "bye", imagine what it is like to have this happen way down deep in a real program. There you are, testing with gusto, and something like this happens. Seemingly, a line just does not execute. It seems the interpreter simply "jumps over" a line of code, and continues execution with the next line down.

Here is the same program again. In this case it seems like the interpreter has gone crazy.

```
10 PRINT "hello"     'a
20 PRINT "and"       'tiny
30 PRINT "bye"       'program
RUN
hello
bye
Syntax error in 0
Ok
Undefined line number in 0
Ok
```

Both of these perplexing puzzles are caused by the same thing. After the remark in the second line ('tiny) there is an extra byte that cannot be seen. It is an FF in hexadecimal, code-255 in decimal. That is why we cannot always believe what we are seeing. In the standard PC-character set, code-255 looks like a space-character. When seen with LIST, it cannot be seen at all.

To duplicate this aberration, put a remark at the end of any line, hold down the Alt-key and index 255 on the numeric key pad. When you let go, that blind byte will be there, believe me.

If the next line is numbered higher than 255, as in the first example above (line 1020), that line will be skipped over. In the second example, with small line numbers, line-30 triggers that crazy pair of error messages about line-0, which does not even exist. Equally crazy, it says Ok, twice.

Oddly enough, it is the not-so-Ok, Ok that may have caused the problem in the first place. That, or some of the other stuff the interpreter dumps on us from time to time.

Most messages like "Syntax error", "Ok", and so on, are output with a trailing, but invisible, 255-byte. When we are editing, there is a risk of picking up that unseen byte inadvertently, while typing over the interpreter's own garbage.

Yes, this happens only on rare occasions: When inserting and deleting characters, and so on, and the line in question just happens to have a trailing remark, and the length of the line is about 80-characters, or so, and the next line down on the

screen was output by the interpreter, and.... But, when this does happen, you will not see it, and when some crazy things happen, you will think you are going crazy because what you can see, you cannot believe. And that is weird.

The moral to the above saga is, at the point in a program where a strange encounter pops up, look at the preceding line. If it has a remark, chop it off. And if this solves your unsolved mystery, get out your Voodoo dolls and pins and concentrate on that faceless character that litters your screen with invisible characters.

To confirm that crazy 255-byte's existence, SAVE the suspected program then use DEBUG or some other tool to look at the file. (Chapter 2 describes what is in a tokenized program file.)

There is another kind of strange encounter that can occur that can really tighten the old rectum: Corrupted program files. Most high quality language products provide at least a modicum safeguard for this one. GW-BASIC makes no attempt whatsoever in this regard. (To be completely fair, it has never been touted as being of particularly high quality, anyway.)

Here is a scenario that can happen. If you have never yet seen something similar, keep bowing down to the East, or whatever it is you do that makes you so lucky.

LOAD "mycrap". LIST. The first hundred lines or so are just what is expected. All of a sudden, garbage hits the fan, from right out of nowhere. This little gotcha is the reason CHKDSK was added to DOS a few years ago. Hopefully you have a recent back-up copy of your favorite program. The alternative to a rewrite can sometimes mean nearly as many hours will have to be spent trying to determine which disk sectors are interspersed with your program files. If you are really lucky, the clusters that contain the rest of your program have not already been overwritten by some interim process.

Far, far worse than this instance is one where you RUN or CHAIN in an ongoing production environment, where the consequence of an unnoticed garbled program just happens to have a few bytes that resemble the BASIC token for something like KILL, or POKE, or PUT, or SHELL, or whatever. An autopsy is hard to do without a cadaver. It is equally hard to do on a disk that has a badly mangled FAT.

A traditional software engineering technique that is often used to give at least some assurance that the sanctity of a file has not been violated is based on a hash-total scheme: Merely a longitudinal summing of the values of the bytes in a file, from one end to the other. By storing this total inside the file itself, it can be compared with another count, done each time that file is loaded. Any difference found can be used to warn somebody that something may be very wrong. Chapter 13 shows some specific tricks that can be used to guard against these strange encounters of the worst kind when writing in BASIC.

One more strange encounter needs to be documented. It portends no system safety risks. For those unaware, however, there is a risk that faulty observations can be made. And that can lead to making incorrect design decisions. None of the manuals in my library mention that TIMER only occasionally produces duplicate answers. When probe coding to determine which techniques are the fastest, be very cognizant that TIMER is a little strange. See this:

```
1000 FOR J=1 TO 50
1010  B=TIMER
1020  FOR I=1 TO 4000
1030  NEXT
1040  PRINT TIMER-B,
1050 NEXT
```

```
 2.023438     1.976563     2.03125      1.984375     1.984375
 2.03125      2.039063     2.03125      1.984375     2.03125
 2.03125      1.96875      1.976563     2.023438     2.03125
 2.039063     2.023438     2.046875     1.984375     1.976563
 2.03125      2.03125      2.023438     2.03125      2.03125
 2.03125      1.984375     1.984375     2.039063     2.03125
 1.984375     2.039063     2.03125      1.976563     1.96875
 1.976563     2.03125      1.976563     2.03125      2.039063
 2.03125      1.976563     1.976563     2.03125      1.984375
 2.03125      2.03125      2.03125      1.984375     1.984375
```

The output shown here was produced by this little program when it was run with GWBASIC.EXE 3.2, DOS 3.3, on an 8088 rated at 8 MHz. Obviously, different machines and different releases of software will produce different results. The TIMER's answers themselves, that is. Even the proportional differences can vary with different hardware. The point to be noted is, a variety of answers will always be produced by TIMER.

To use TIMER as a tool for doing performance studies, tests should be repeated at least fifty times, or so. An average of those results will be reasonably useful, although, still not precise.

For those that are curious, here is another output listing from this same program, run on the same machine, under DOS 3.3 but, as an in-memory compiled program generated by the QuickBASIC 2.2 compiler with the DEBUG switch on, and with all event trapping switches off.

2.914063	2.90625	2.914063	2.914063	2.914063
2.851563	2.851563	2.914063	2.859375	2.859375
2.90625	2.914063	2.90625	2.914063	2.90625
2.859375	2.859375	2.921875	2.851563	2.84375
2.914063	2.90625	2.921875	2.867188	2.90625
2.90625	2.851563	2.90625	2.914063	2.859375
2.859375	2.90625	2.859375	2.851563	2.84375
2.914063	2.921875	2.90625	2.914063	2.90625
2.851563	2.90625	2.914063	2.859375	2.859375
2.914063	2.90625	2.851563	2.90625	2.90625

One slight change was made to the program before it was run for this listing. Line 1020 used a limit of 4000 for the loop that does nothing when it ran with GW-BASIC. For the QuickBASIC run, that limit was set to 8000.

It is interesting to note also, to do nothing twice as many times took longer with the compiled program vs. the interpreted one. One is tempted to think from this test that compiled programs run nearly twice as fast as interpreted ones. Almost. Maybe. Sometimes.

More than once a strange encounter has been experienced because it was assumed that a compiled program would run faster than an interpreted one. Usually they do, that is true, but not always very much faster. Chapter 11 suggests several considerations needed when designing programs in BASIC, including some factors that need to be considered when choosing between GW-BASIC and QuickBASIC.

Those in the know are always skeptical of magazine ads. It is strange that anyone can believe some claims. Yet, even with what we know, we can still have some strange experiences. It would not happen as often, perhaps, if our systems manuals were more informative than they are.

The micro boom was responsible for many cultural changes. One that is never alluded to by industry scribes is typified by that very reluctance: The shifting of that fine line that demarcates honesty.

All software has bugs. Professionals on the big machines have always known that. When someone pays a million or two for a computer, they expect, and get, truthful systems documentation.

For decades all new software releases were delivered with a list of "restrictions", up front. An honest admission that not everything worked as intended, and explicit advice to not use this or that feature.

Such admissions did not hinder the growth of that industry. In fact, it was essential to mere survival. Large data processing operations that cost thousands of dollars a day to run could ill afford to find out what worked and what did not by trial and error.

In that world, decisions about which vendor's products to buy depended on those with technical acumen. Experts confronted experts across the table when the buyers and the sellers sat down to wheel and deal. Hogwash and technical incompetence were neither one tolerated--any of either could kill a deal.

The PC industry is a different world in a lot of ways. Most purchases are made by end-use consumers that have virtually no technical expertise. My loathing of general allegations like this does not dissuade me in this case. It is inexplicable to me how some of what we have to put up with manages to flourish in the market place, year after year, save to presume that the buying public is gullible.

Unless Nader's Raiders or the folks that champion causes like truth in lending laws come to the fore, it appears that we will have to contend with what we have: The companies with the most bucks behind them set the stage on which we have to act.

Hopefully my little show here will enhance your productions. It is certainly possible that I may be found guilty of an error in commission, but recognize my honest attempt to not be accused of dishonesty by reason of omission.

May some of your strange encounters in the future seem not so strange after all, after seeing what I found to be strange, at

one time or another.

Chapter 10 = STYLE

"Programming involves both art and science."

That observation does have a nice philosophical ring to it, and that may be one of the reasons we hear it echoed from time to time. Admittedly, it falls softly on the ears of my ego, too.

There is another observation about programming that ought to be made, but it is seldom seen in print; it has a crap-cutter edge to it, useful for soul searching, but it scratches eyes, ears, and egos.

"Programming involves both intuition and luck."

A smooth utterance in my favorite Webster's defines intuition as: "The power of knowing, or the knowledge obtained, without recourse to inference or reasoning...." Fifty some-odd pages later, the definition of luck begins with: "That which happens to one seemingly by chance...."

Eloquent word working can sometimes make intuition palatable. Luck is a tougher pill to peddle to programmers that embrace "logical reasoning ability" as a personal virtue.

A respected teacher once helped me grasp how intuition impacts what we do: "Experience causes us to mentally tag that which works well, and to log that which does not with a different type of tag. The monotony of the mundane in our daily labors soon causes these memories to form mental habit patterns by which we intuit how to do similar things, with no strain on the brain."

Hopefully, the good and the bad were tagged correctly when first filed in our memory banks. This thought is mine, on reflection, and is one of the reasons for my contention that luck is a factor in the overall definition of what makes all programmers tick.

Remembering further remarks of my esteemed teacher, she also told me that what she could not teach was experience. The best she could do was to try to instill good habits early on. Her definition of good, was, "... a distillation of experience of those who already had it". Once more my thoughts conjure up an element of luck. Hopefully the one distilling the brew uses the right recipe, and the ingredients that are used have come

from a preferred source.

Presumably you have been programming long enough that your own habits have already developed strong roots. This chapter is not meant to be presumptuous; in no way do I presume my habits are the best, nor even, any better than someone else's. By dissecting my habits of style on paper, however, along with an honest attempt to recollect their roots, perhaps some will be useful seasoning to add to your own brew.

A popular theme of the seventies was "structured programming". Many books can be found on that subject, with many definitions of what it is, and how to do it. None attempt to use BASIC to illustrate their concepts. Probably because of GOTO.

One nickname for structured programming is GOTO-less coding. That is awkward to do in BASIC. In fact, it is a bad idea in many cases, if we want to run in the fast lane.

Fanatics from the structured programming schools have caused a lot of inbreeding among programming languages. The concept of "block structures" was grafted onto BASIC, even, back about the time Pascal was all the rage on the West Coast. Knowing a tad of that history helps us see why we have, what we have.

Pascal was invented by N. Wirth, in Germany; it showed up first in the U.S. at UCSD in California. Professor Wirth gave us what he thought would be a better "teaching language". When Pascal hit the micros, it was initially an interpreted language, similar to how BASIC worked in those machines. But Pascal did have "blocking statements", and that was popular, from coast to coast. So much so that those whose first language of love was Pascal had very little empathy for old-timers that had courted many languages in our youth. None could understand how anybody could ever have dated a dog like BASIC.

The idea of "block structures" as an inherent feature of a language came from ALGOL, originally. It was supposed to be an international programming language, equally suitable for documentation purposes. It has a strong flavor of FORTRAN about it. So does PL/1, which has about everything, including an admixture of FORTRAN and COBOL. ALGOL and PL/1 both came out of IBM labs. IBM was (and still is) a big name on college campuses. General Electric was too, although GE computers were never manufactured in as many numbers.

BASIC began life on a GE computer at Dartmouth College; it was intended to be useful as a poor man's version of FORTRAN. It ran on only very big systems that offered time-sharing via telephone hook-ups. A user did not have to own a computer; he merely needed a cheap terminal and pay charges for system use only while on-line. And pay the phone bill, of course.

A primary contention of the inventors of BASIC--Professors Kemeny and Kurtz--was that, it could easily be learned by non-computer-oriented mathematicians and engineers. "Beginners Algebraic Symbol Interpreter Compiler" was a contrived phrase, probably, to justify an acronym that resembled an English word. That not-so-subtle double entendre caused BASIC to grow up with an inferiority complex. Beginners, in the beginning, however, were mostly well-educated college graduates.

The micro boom moved BASIC lower on the social scale. By the time the word beginners included grade school students, it was considered as demeaning by upper class folks. Many new-hire programmers considered BASIC unbecoming to their sense of professional dignity. They simply ignored the fact that, many of us had already earned substantial salaries writing large scale business applications in BASIC, often requiring levels of expertise some of those neophytes would never attain in any language.

Meanwhile, visionaries at the helm of the companies that made our machinery ignored campus love affairs between students and programming languages. It seemed to them, if BASIC could have a common definition to all, they could all sell a lot of cheap iron. A million of anything at a dollar was almost as good as a single, any one thing that cost a million per copy.

Customers that could afford to spend millions were expensive to court. To sell to those who could afford only small capital outlays, they needed a simple language. BASIC was touted for a number of years as being good for first-time users--it was invented for beginners, after all. Far more importantly, it could be built into machines with small memories. Initially, anyway.

The half-hearted attempt by ANSI to standardize BASIC probably hurt us more than it helped. The heavy-weight vendors like IBM, NCR, and DEC insisted that the ultimate language should

include features peculiar to their own dialects. My input to the X3J standards committee (as an NCR contributor) argued vehemently against things like OPTION BASE and RANDOMIZE. As feared, those that later bragged they fully conformed to the standard also gave us bigger, more cumbersome, and slower running interpreters.

Now we have BEEP and SOUND. An obvious redundancy. Things like SOUND, PLAY, and DRAW had not even been invented when the standard was written. GET and PUT were infants. Fortunately, ANSI shied away from trying to standardize any I/O commands beyond the simpler ones like INPUT and PRINT.

Having watched it happen, it is no mystery to me why most of what is in "standard BASIC" are the things we use the least. That which rigidly conforms to the standard is also that which is most often inefficient, slow, and many times, pragmatically useless. That old saw about a camel being a horse put together by a committee is typified by BASIC as we see it today. Dummy arguments needed in functions like POS(1) truly are dumb.

It is odd that a language which was originally compiler based, and thought by many as dumb, wound up as the staff of life in interpreted form, burned into the memory of what so many people think the word computer itself means.

BASIC imparted intelligence to machines that had an IQ of zero. PCs still resemble terminals, and are often used as such, but most are powerful enough today that they can be programmed in virtually any language, with compile speeds approaching, and sometimes surpassing, that of their granddaddies.

Although BASIC was originally a perverse dialect of FORTRAN, its early compilers were actually heavier than those for its big brother. Because BASIC had a line-at-a-time orientation, it was easily adapted into micros with small memories using a software interpreter that "translated" what a program was supposed to do, as it ran. Now the wheel has gone nearly full circle.

BASIC has evolved to the point that it looks more like its cousins, than like its grandfather. Even to the point that, line numbers are no longer needed when using compilers like QuickBASIC. Branching can now be done to "line names", which are managed by the compiler in the same fashion as is done for variable names. (The original idea of "paragraph names" is a

hallmark of languages like COBOL. GOTO, to a line-name, goes all the way back to stone-age assembler languages that roamed the earth about the middle of this century.)

This constant migration of structural concepts among languages has caused a lot of side affects, not all of which are truly beneficial: Especially for some things, like WHILE and WEND, which experienced ramifications similar to FOR and NEXT, in interpreted languages like GW-BASIC. (FOR and NEXT suffered first from the ANSI impact. Having paid that price, it was relatively easy to inject WHILE and WEND, in an attempt to rejuvenate enthusiasm for BASIC. At one time, some feared it was in jeopardy of being supplanted by Pascal.)

Today, nearly all languages have a WHILE and WEND counterpart. Most also have some form or other of CASE-statements. GW-BASIC does not have CASE yet (thankfully), but its QuickBASIC sibling does. None of these additions were really needed in any BASIC, especially not in interpreted BASIC. What follows are a few of my opinions on how much we should allow "structured programming concepts" to influence our style of coding in this language.

WHILE and WEND are useful, and efficient, if used in a very limited fashion. Rule number one: Keep them close together. On the same line, preferably, as in this example.

```
I = 0 : WHILE I<1000:I=I+1:WEND
```

When the interpreter bumps into WHILE, it immediately takes a trip, looking for WEND. Then it sets up stack-pointers to keep track of where the block begins, and ends, and the "if" needed to terminate the loop. Then it begins doing whatever it is supposed to do inside the loop, if the implicit-if has not already been satisfied.

This same thing is done today with FOR/NEXT loops. This was not always so. Before ANSI got involved, a FOR/NEXT loop would always run at least once in most dialects of BASIC. A basic concept of source-code interpreters was to ramble along, doing whatever is encountered next, one step at time. In those days, it did not matter where NEXT was; the interpreter simply made a mental note when it hit FOR, presuming that it would eventually run into NEXT and would remember where it had seen FOR, before.

Today FOR/NEXT works differently. Because nothing inside the loop is done until the bottom of the block has been located, it

is possible to not do anything, i.e., to ignore everything inside the loop if the TO-limit is less than the FOR starting argument.

When the interpreter examines FOR and realizes that nothing is to be done, it simply runs ahead, looking for NEXT, then keeps right on trucking from that point on. Which is why you can have all kinds of errors inside FOR/NEXT and WHILE/WEND loops that go unnoticed by the interpreter; errors that are always flagged by a compiler, even if what is inside a loop may never get a chance to perform. (Chapter 12 suggests how to use the compiler as a programming tool, even for programs that are going to be used only in interpreted form. And vice versa.)

See the difference. A compiler examines everything, in every line of a program before an "object program" is produced, i.e., the program that will actually be in memory at run-time. An interpreter loads all of a "source program" into memory, but it has no idea what it is going to run into. It simply takes things as they come, one step at a time. At least that was an original concept of a source-code interpreter. Today, the side excursions that GW-BASIC takes when it hits WHILE, or FOR, are an expensive contrivance that can impact performance if we are not mindful of how the interpreter does it.

Because a compiler gets a preview of a program, and it can "hard code" where things are, like block boundaries, in the run-time code, it does not matter how big a block is. Or how many there are, or how deeply they are nested.

Rule number two for interpreted BASIC: No nesting. At least, keep it to the absolute minimum. And keep the innermost blocks as short as possible. Remember, for every iteration of an outer loop, the depth of an inside loop must be determined all over again. Every time the interpreter takes that trip, you pay for its vacation. All the while it is stumbling along, scanning each line, looking for the end of a block, your mission is in limbo.

Another reason for using "blocking statements" conservatively is because the interpreter momentarily forgets what you are doing while it is off sightseeing. See this, which is a short loop coded with one statement per line.

```
1000  WHILE CVI(Q$) = 0
1010   MID$(Q$,1) = INKEY$
```

```
1020  WEND
```

This is good technique, but, it can also cause some funny experiences. Sometimes BREAK will cause an ERR = 8. This happens if the interrupt is sensed while the interpreter is fooling around, pretending it is a compiler, as opposed to actually executing statements inside the loop.

With an awareness of what goes on inside the interpreter, it can be seen why GOTO is a preferable alternative to using WHILE and WEND. The performance advantage is proportional to the frequency of use of a specific piece of logic. A "block" that is executed only once need not be a concern, but, see why

```
2000 GET 1 : IF LOC(1)<LOF(1) AND R$<>" " THEN 2000
```

is a better programming style when lines like this are buried deep inside loops that must be executed thousands of times. At a point, when to use which becomes an intuitive decision.

There are times when we should take our brain off autopilot and revert to manual reasoning. When coding inside a loop, that is nested inside a loop, that is inside another, and so on, performance considerations may dictate that we ought to return to old fashioned coding styles. A girdle may improve appearance, but it is seldom comfortable.

As an element of style, the issue of naming FOR-variables after every NEXT can be argued about forever. My habit is still, to always use a generic NEXT, because of the performance issues enumerated in Chapter 3.

Many will contend that NEXT should always rename the control variable that was used with its FOR as a visual aid. Perhaps that is a good idea when FOR and NEXT are miles apart, and when considerable nesting has been done. The visual-aid argument is weak for my loops, where the distance between FOR and NEXT is kept as short as possible, and very, very little nesting is done.

Once again, coding conventions should be decided upon, on a global basis. We should not adopt "rules" one at a time.

Structured programming preachers had a lot of followers that did not fully understand the religion. Some of those souls

spread the gospel so well, interpreter users can end up with a heavy cross to bear--a burden that can be lightened if we heed our instincts and do our own thing, even when it is not popular with the multitudes.

Not all of the gospel should be ignored, however. Some of it is very good advice. One of their favorite tenets is often called the "single entry, single exit" rule. It is a favorite of mine now, adhered to with fervor and zeal. No longer can we do things like this:

```
1000 GOTO 2000
1010 LOCATE 12,1 : PRINT I;J;     'a commonly needed function
1020 NEXT : RETURN                'a single, catch-all NEXT

2000 FOR I = 1 TO 10              'outside loop
2010 GOSUB 1010
2020 FOR J = 1 TO 10              'inside loop
2030 GOSUB 1010
2040 ....
```

It is nice that we are no longer so memory-bound that we have to resort to tricks like that above, just to save a few bytes. It is not so good that zealots have been allowed to pass along a cost factor for what our natural inclinations would likely have been, anyway. All "loop-blocks", in any program of mine, have but a single exit.

The end must follow the beginning, physically, whether we like it or not. Because the interpreter is trying hard to mimic a compiler, it scans in a forward direction only when looking for the bottom of a block. To be able to do it as fast as it can, it concentrates strictly on finding a WEND or NEXT that it supposes is further down the page. In its head-long rush it ignores everything but what it is looking for, including GOTO and its kin.

As an adopted discipline, an element of my style goes beyond what is dictated. Conditional tests may be done on any line within a block of procedural statements but, if they want out early, they must force the "if" that will satisfy a NEXT or WEND, then branch to that line at the bottom of the block. Never, ever, GOTO out of the middle of a FOR/NEXT or WHILE/WEND structure. As a matter of habit, do it thus:

```
1000 FOR I = 1 TO LEN(X$) : E = I
1010  IF MID$(X$,I,1) = " " THEN I = LEN(X$) : GOTO 1090
```

```
  ....
  1090 NEXT
```

Another of my natural inclinations has always been to arrange all chunks of a program into functional blocks: Procedural tasks are organized as function-oriented subroutines. Any GOSUB that calls a task must always be aimed at the same, first line of that block. The last line of all tasks contains nothing but RETURN, and, it is the only line that is allowed to do a RETURN. Thus, by my definition, my style does resemble "structured programming", as exhibited in this example:

```
  1950 'sort
  1960 FOR E = -1 TO 0
  1970  FOR I = F TO L-1
  1980   IF A(I)>A(I+1) THEN SWAP A(I),A(I+1) : L = I
  1990  NEXT
  2000  E = L<I
  2010 NEXT
  2020 RETURN
```

The "name" of a task is a REM (coded with an apostrophe for aesthetic reasons) that helps me remember what that subroutine is for. That short identifier is the only thing on the first line of any subroutine; it is the line that any using GOSUB must be aimed at. The reason the remark-name must be kept short is because it is dead code, remember. (As described in Chapter 2, when the interpreter hits a remark, it has to bump along, one byte at time, until it finds the start of the next line.)

So, single entry, single exit: Subroutines begin on the first line and end on the last. Because the entry point is a no-op line, and the only way out is the final RETURN, they always have a logical, physical, and visual, block-like appearance.

Because they look like blocks, they are easily seen as such when scrolling or strolling through the code. Because each begins with a short nickname, it is easy to remember which block is for what.

A physical advantage to this scheme is that changes can be made easier. Something that needs to be added up front can be inserted just after the do-nothing name line. Last-act changes can be added where RETURN was, and a new RETURN can follow the addition. Because all internal early-exits were always aimed

at this single exit point, there is no need to go back through the block to ensure nobody skips task clean-up chores.

The logical advantage to the one-front-one-back-door idea is important for real blocks--WHILE and WEND, FOR and NEXT--and for subroutine-blocks. If always done as a matter of habit, stack overflow errors should never happen.

Without a doubt, the meanest bug you can hatch is the one that causes an "Out of memory" message (ERR = 7) when it is because the interpreter's "stack" has been blown. This stinker pops up most often because of a bad branch somewhere; somebody escaped from a block without going through the bottom. Chapter 12 describes some methods for finding the line that is guilty of this sin (and how to avoid use of the two most useless words in BASIC, e.g., TRON and TROFF).

When GOSUB, WHILE, FOR, or any of the trap-triggers like ERROR are encountered, the interpreter does a PUSH of pointers onto its stack. RETURN, WEND, NEXT, RESUME, and similar statements do POP for a corresponding number of times. If more pushing than popping goes on, at some point the stack will be full and the whole show screeches to a halt. The most comprehensive error handler cannot cope with this.

Because ON ERROR has to use the stack, if it is full it cannot remember where to RESUME to, even if that seems like a good idea. It is probably better to forget the whole thing and just crash, as gracefully as possible. Any program that blows the interpreter's stack is a borderline psychopath in dire need of diagnosis and treatment. It should not be allowed to run amok and cause someone irreparable harm.

Whether thought of as style, technique, method, or whatever, this seems an opportune point to mention some closely related bad habits to avoid. Emphatically, in my school, we never do a RETURN to-a-line-number, or a RESUME to-a-line-number.

It is amazing that RETURN-number showed up in BASIC in the same era that WHILE and WEND creeped in. One seems to me, to be a direct contradiction of the philosophy of the other. Granted, it does take a little effort to design for event-trapping logic such that, after the event, the continuity of whatever was happening previously can continue so as to not circumvent the single-exit rule for the bottom of the block that was in motion

when a trap does occur.

There is nothing wrong at all, of course, with doing a GOSUB from within one block, to another block. This presumes, of course, a called-block will always end in RETURN, bringing control back to the block from which the GOSUB was done, which will allow that block to exit through its own back door. Doing a RETURN to a specific line number is anathema to me, and, to all advocates of structured programming.

This advice has greater significance for ON ERROR GOTO. To not RESUME-number, we have to RESUME NEXT. There is an old bug in BASIC that looks like a centipede because it has so many legs. It is defined at length in Chapter 9, but a simple description is all that is needed here: Just where NEXT is, is not easily reckoned sometimes.

Although the manuals say RESUME NEXT will cause execution to continue at the next statement following the one that triggered the ON ERROR jump, the interpreter often loses track of where it was when an error was encountered in complex or compound conditional expression (e.g., those using THEN and ELSE).

So, our coding style must cover for this old bug, just so we can stay alive in this business. Which has nothing to do with history, religion, or getting an A in school. Several rules are necessary for survival; a couple more are necessary to keep your bugs from interbreeding with those buried in the interpreter.

Rule-1 in the book of ON ERROR: One error-handler. It too should be constructed as a "block". Its single-entry point is defined once, by a single ON ERROR, up front in the program. The bottom of an error handler is a single RESUME NEXT.

If ON ERROR is turned off (with a zero), or reinitialized to this same address, or some other, no RESUME can be done other than to a specific line. If you RESUME to a line you may cause the interpreter to blow its stack at some point. Which will likely cause you to blow your stack immediately thereafter.

As the manuals say, but without elaboration, only one ERROR can be coped with at any given point in time. If that trap is taken--which is conceptually like a GOSUB--a second error cannot be tolerated until a RESUME is done. In software engineering parlance: An error trap (and a subroutine) cannot be recursive.

A call to one's self is not permitted in BASIC.

Rule-2 in this book: Do as little as possible while trapped inside an error handler, and get out of there as quickly as you can. Set a flag, or something, and RESUME NEXT. Let the guy who triggered the error take care of the situation. This does mean, of course, that procedures elsewhere must anticipate that a return from the error handler can come back waving a flag.

Rule-3 should maybe have been the first rule: Do not do any I/O to mechanical devices while inside an error handler. (My ideas about logging errors is covered in Chapter 11 along with other design issues. Here, we are contending with the matter of how our coding style has to anticipate that errors can occur anytime.)

The simpler causes of errors--our coding mistakes--can be found and eliminated (eventually). Because we cannot have blind faith in those over which we have little control--like DOS, and even, high priced disk drives--we have to anticipate that they do malfunction sometimes. It is best to not allow them an opportunity to confuse us with another error while trying to cope with the one that caused us to be trapped inside an error handler in the first place.

Once our code is fairly clean--only the naive believe they can write bug-free programs--the most likely trip to an error trap will be triggered while doing I/O to mechanical devices. It can happen anytime. So, do not do any input or output after ELSE. In fact, if it is done after THEN, do not do an ELSE on that same line. This rule should be followed throughout, in any program that has an error handler that uses RESUME NEXT.

Because the interpreter gets confused about where next is, on any line that has THEN and ELSE in it, the above rule must also prohibit deliberately provoking errors with ERROR-number on lines that include THEN and ELSE.

For a number of years my programs often used ERROR-code to tell the operator about keying mistakes. The error handler had all the overhead for doing BEEP, LOCATE, PRINT, and the like, and, displayed selected messages from a tank, based on the value of the number used with ERROR. (In the strictest sense, output to the tube is I/O, but "Device I/O error" and the like were rare enough that it seemed practical to bend the rule about not doing I/O in an error handler, for a non-mechanical monitor.)

Two things have caused me to forsake the use of ERROR-number as a cheap substitute for GOSUB. The rigid self-discipline needed to avoid an unpredictable return from RESUME NEXT became tiresome. Trying to remember where an ERROR would come back to, in conditional expressions that involved both THEN and ELSE was harder than the simpler alternative of assigning a code to a variable, then doing a GOSUB to a general-purpose "message subroutine". The idea that monitor I/O errors were not likely, is not as valid as it used to be, either.

Output to a monitor, using conventional BASIC commands like PRINT, invokes BIOS calls. (Chapter 7 dwells on this.) The continued proliferation of new types of monitors and adapters increases the risk that sooner or later one of them, or DOS, will change its mind about the signals it sends back to BASIC indicating the results of output requests.

Even if we suppose that "Device I/O" errors are still unlikely on a monitor, there is another overall design issue that must be considered. Once upon a time, all that a general purpose message routine had to remember, and restore, was where the cursor was before the jump occurred. Today we have to contend with color, cursor-on or cursor-off, the cursor's size, display pages, and on and on. Today it is usually simpler to move this overhead to specialized subroutines geared to different screen modes. Error message output to a monitor in an error handler should be limited to "emergencies" as a matter of habit.

Having reasoned why our coding style must be cognizant of the risk of errors occurring between THEN and ELSE, the use of IF itself needs some thought when speed is worth worrying about. IF is about the slowest thing you can do in interpreted BASIC. Almost any alternative is usually faster.

Heeding this advice, see also why WHILE should not use multiple implicit "ifs". WHILE A>B AND C<D AND E>F is the equivalent of doing the same thing with an IF, each time WEND is encountered. Reconsider this old-fashioned alternative:

```
1000 IF A>B THEN 1040
1010 IF C<D THEN 1040
1020 IF E>F THEN 1040
1030 ....              'exception logic
1040 ....              'otherwise
```

The authors of some textbooks encourage us to stack IF-lines like this so that the condition most likely to occur is tested first, the next most likely second, and so on. This author agrees with the reasonableness of that thinking, but offers a suggestion to consider as superior to that: Do not use IF at all when there is a practical alternative.

One such alternative is ON GOTO (or ON GOSUB). This has always been to BASIC, conceptually, what CASE is to other languages. Hence my inference that we should vote against CASE in BASIC.

CASE originated in languages that had nothing comparable to ON GOTO. Chances are, if it was added to BASIC, it too would not be very efficient because of the artificial contrivances that are patched into the interpreter when it tries to adopt any type of structure that is foreign to what was, and still is fundamentally, a line-at-a-time language.

The following example is a rewrite of the one shown earlier. This one uses ON GOTO as an alternative to IF.

```
1000 ON ABS(A>B AND C<D AND E>F) GOTO 1020
1010 ....                                   'exception logic
1020 ....                                   'otherwise
```

Not only is this shorter, it is a faster alternative when the various IF-conditions enjoy an approximately equal chance of happening. It is the faster alternative only when no heavy arithmetic is needed in the ON-expression, however. Contrast this with the following one that produces different addresses for each of the conditions being tested:

```
ON ABS(1*(A>B AND C<D)+2*(A<B AND C>D)) GOTO 2000,3000
```

which is equivalent to:

```
IF A>B AND C<D THEN 2000
IF A<B AND C>D THEN 3000
```

As with any advice from a book, this has to be eyeballed for what it is worth in given situations. Before your thoughts turn argumentative: Yes, IF and THEN are easier to read than ON la-de-da. My thoughts about how coding style and program maintenance issues can be judiciously balanced are offered a little later. My admonishment here is, adopt nothing out of context of the whole of this subject to avoid being labeled

an extremist.

A good example of extremism was APL. It had no IF. (APL came out of Harvard about 1962; was all the rage for a while on the old IBM 360 computers; it has steadily declined in popularity since about 1975.) Virtually everything done in APL has to be accomplished with algebra-like expressions. George Boole and Blaise Pascal would have loved it.

APL was good for machines, but it was hard on programmer heads. In my youth it was fun to develop lines like

```
((A>B)/'G'),((A=B)/'E'),(A<B)/'L'
```

so that G, E, or L would print, to show the relationship of A to B as being Greater, Equal, or Less. And this is a simple example. Compare it to this one for evaluating an algebraic polynomial:

```
3 + (2 x Y) + (9 x Y * 2) + 4 x Y * 3
```

Believe me, if you wrote twenty lines in succession like this, in the morning, and did not notice immediately that one had a mistake in it, you could get a bad migraine trying to find the error that afternoon. APL was, is, and shall forever be (we hope) the most cryptic of all languages ever foisted upon us. Even FORTH was not that obtuse. It too is now a dead language.

Now we come to the issue of living in glass houses and throwing bricks. There is a difference between a language itself being cryptic, and our personal choice to sometimes write cryptically in a language that does not insist upon it. See this line out of one of my own tricks displayed in Chapter 14:

```
3240 E=E*(VAL(LEFT$(X,2))<13)*SGN(VAL(LEFT$(X,2)))
```

Unlike in APL, we can decide when to use IF in BASIC. And we can deliberate about when, or when not to be cryptic. In this line (3240) a "conditional" expression is used to generate a "flag" in a numeric variable. Obviously, seen out of context, it is not obvious at all what its purpose is. Programmers that know BASIC well can easily see what this line will do, but not why, nor to whom.

Occasionally we still see comments in our junk mail that BASIC

is an English-like language. The non-programmer scribes that dump this crap on the public have never tried to "read" one of my programs. In fact, any program, in any language that can do more than "See Spot run", is impossible to read one line at a time.

Even COBOL, which is extremism in the opposite direction of APL, is not always as "readable" as its proponents vehemently argue, in spite of its determination to mimic English. You cannot read one paragraph of a COBOL program, alone, and infer its reason for being, without reading the whole book.

Argumentative or not, the above paragraphs are offered as the underlying preamble for my attitude that my programming style does produce programs that can be cost-effectively maintained. Which is THE issue by which to argue using tricky techniques for conserving space or increasing performance vs. those that are (somewhat) easier to read, for those of us earning our daily bread as programmers.

Teachers, preachers, authors, text books, and technical tomes have to try to communicate to the masses. What a real program of mine looks like, internally, is privy to a very small group of people; often as not, only three: Me, myself, and I.

The sort routine shown earlier is recoded here, as it would actually appear in one of my programs, as a model of my style, for what I call freeze-dried code.

```
1950 'sort
1960 FOR E=-1 TO 0:FOR I=F TO L-1
1970 IF A(I)>A(I+1) THEN SWAP A(I),A(I+1):L=I
1980 NEXT:E=L<1:NEXT
1990 RETURN
```

The name line is short, and, uses lower case letters to make it easier to spot when scanning listings or scrolling on the tube. Notice there are no other remarks. They are not needed by me, or another programmer wandering around in my code. That is my assumption.

Freeze-dried code has no need to be artistic or pretty. It either works, or it does not. Once debugged, it is highly unlikely this routine will ever have to be changed, hence, it should be coded so as to be as efficient as possible. The more compact it is, the better. Conceptually, freeze-dried code is

like an "intrinsic function". In this example, if BASIC had a SORT verb, we would never see these lines, nor worry about what they look like. Our own-code substitute should strive to be as small and as fast as possible. Effectively, GOSUB-number can be used as a substitute for your own extensions to the basic, BASIC language.

To minimize future maintenance efforts, areas in a program that are likely to have to be modified someday deserve a different attitude about style: They should be easily found, easy to read, and illuminated with remarks.

Some of my habits make it easy to find things, like using a one-word delimiter at the top and the bottom of a block. Most of the lines inside the blocks are long, multistatement lines. No single line is ever longer than the width of the screen, however. All line numbers are always 4-digits. (My programs always begin with line 1000 and are incremented by ten from there on, with no gaps. Some additional reasons for this are cited in Chapter 12.)

A picture is worth a thousand words, so goes an old cliche. The "image" of a block can be quickly seen, by reason of my habits of style. Once the picture that is wanted is on the screen, concentration can then focus on a word search.

Notice my reluctance to use the word module. Text books on structured programming love that word, but few use a common definition of what it means. My definition: Not more than a screen full. LIST is tiresome. The cheapest word-processing programs can scroll text forwards, and backwards. The editor in GW-BASIC cannot scroll in either direction.

Dense code runs faster, and, it enhances productivity. After the labor of getting a chunk of logic on the screen, we want to see all we can. To relate to a part that is not visible, our mind has to shift gears to fetch another picture. By then, it is hard to remember how that information corresponds to what we were looking at just moments ago.

With this pair of motives in mind--optimum performance of man and machine--a few more of my habits need to be listed.

Restrict all lines to a maximum length of 80: The blanks

following lines that do a wrap-around contain little useful information. Continuity of perception is disrupted at the point of overflow, and, when scanning down the left margin looking for line numbers.

Omit all "optional" syntax: In CLOSE #1, for example, the pound sign is not needed by man or machine.

Condense conditional expressions: IF I-1 THEN is shorter and faster than IF I<>1 THEN.

Use DEF-type, and omit variable-typing appendages: Having said DEFSTR M-Z up front, all of those tiresome dollar signs can be left off of all string expressions, and save a lot of space. And shift-key usage.

Use short variable names: One or two characters is the rule.

Way back when, a variable name in BASIC could only have one or two characters, plus a data-typing appendage. And it was that way for a lot of years. At a point, names could be longer, but only the first two characters were used to discriminate between names. Today we can have names up to forty characters long, and all of them are used in doing name comparisons. Big deal.

Before they gave us so much, and slowed us all down whether we liked it or not, we had already learned to live with what we had. Those methods are just as viable today. Conversely, the advantage to be gained by longer names is not enough to make this old dog learn new habits. Here are some of my old tricks of the trade, and why change is resisted.

DEFINT C-L : DEFSTR M-Z : DEFDBL A

This is done once, only, at the beginning of any program. It is extremely rare that this allocation of the alphabet ever has to be different. The unstated B is, by default, for single precision variables. Of all of the expressions in a program, few need to use floating point variables. A and B, followed by other letters, or numbers, will provide for up to eighty of each type. (The full range for either is 164; 82 simple names and 82 arrays, including the use of the single letter itself, and allowing for one name to use a period as a second letter.)

Most of the "computing" that is done in all BASIC programs is

for the benefit of counting and making decisions. Most of that can be done with integers, and it is definitely faster to do so. Hence, the use of C-L for integer names.

Data file processing, especially, uses a lot of strings. The M-Z half of the alphabet is sufficient for that, and all other string manipulation tasks whether file-related or not.

Once adopted as a habit, a consistent allocation in the use of the alphabet makes it easy to see and read names without any trailing data-type appendage. It also causes us to develop strong preferences for the use of certain names in a similar way, for a long time, in a lot of programs.

A name beginning with the letter Q, for example, always has something to with keyboard activity in my programs (e.g., Q is for Query). Similarly, my first choice for a FOR/NEXT loop is always the letter I, stemming from thousands of lines of code written in FORTRAN. (In that language data typing was enforced upon us; early FORTRAN compilers gave us no choice; certain letters were pre-designated as being the ones we had to use to reference certain data types.)

Another benefit of short names, used the same way over a long period of time, goes way beyond the obvious advantage of our ability to easily remember what they are used for: When merging code from one program into another, far less work is involved than would be the case if all of the names had to be changed. And, this also reduces the risk of editing mistakes.

Chapter 3 presents arguments about declaring all variable names up front in a program. Add to that advice, this is also the place to make notes about what variables are used for. Here is a short segment of the front end from such a program:

```
1190   I=0:J=0:K=0:D=0:E=0:F=0:G=0:H=0:A=0:B=0    'local
1200   L=CSRLIN:C=POS(0)                          'global
1210   BM=&HB800 'BaseMonitor
1220   CM=0:CQ=0:CP=15 'CursMon:CursQkey:CursPrt
1230   C4=0:L4=0 'Col4:Line4 limits
1240   DQ=0:EQ=0:GQ=1:KQ=0 'DoQ:EditQ:GetQ:KeyQ
1250   ME="GenFont2.E00" 'MaskEdit
1260   MH="GenFont2.H01" 'MaskHelp
1270   Q1=CHR$(0):Q2=MKI$(0) 'Q1key:Q2key
```

Self discipline can produce many side benefits from habits like these. While coding, if a new name has to be invented, look first

at the start-up names list to preclude accidental conflicts. By always adding new names to this list before they are placed into use, they will not be forgotten, neither the names, nor what they are used for. This scheme also helps keep the list short. The possibilities for using some variables at different times for different purposes can be more readily seen in this way.

As a general rule, single character names are always local; two characters are used for global names. Meaning: After a GOSUB, do not depend on the contents of single-letter variables. Names with two characters have values that are global in nature, either in the sense of the program overall, or among just a few routines, even. In some cases, as in the example above, C and L are used throughout as global names by this particular program. It is one that uses LOCATE a lot. To keep those lines short, these two single-letter names were declared global in this program as an exception to the usual rule about single character names being subject to capricious use inside any subroutine.

One more habit needs to be mentioned as a matter of style. My overall attitude about remarks can be seen in many places. Those shown with variables in the earlier example are typical: Short, terse, and somewhat cryptic. Given my habits, however, and an awareness of what that program does, they are as useful to me as they would be if they were each a paragraph long.

Down in the bowels of a program proper, my remarks tend to be scarce. They are usually used for annotating logic-flow, as opposed to explaining how something works. Knowing BASIC, it is not too hard to decipher mechanics, but it is not so easy to see why a GOSUB is jumping off to somewhere. The following one-line example solves this type of problem at a small cost.

```
3440 IF I THEN GOSUB 1030:GOSUB 1110 'sort:save
```

This line is from a program that had a subroutine called "sort" and another called "save". A short REM coded on the tail end of any line that says GOSUB should duplicate the same nickname that is at the top of a called subroutine. This habit, adhered to with Teutonic discipline makes it easy to "read" a program.

Most often when we are looking at old code--even that written only a few days ago--we are usually trying to track its logical flow. Once debugged, the grit in the middle of expressions is like sand when what we are really looking for are boulders. Our blunders at this point are more likely to be errors in logic, in

linking major blocks together, rather than picky errors inside the blocks themselves. (Chapter 11 provides some ideas about how to decide what blocks are needed. Chapter 12 describes in more detail how and when to best build, and debug, the pieces of a program.)

This chapter should not be read as an attempt to thwart the advice of wiser heads than mine. Nor should any motive be perceived to contradict what may be considered by many as "good programming practices."

Pretty code vs. dense code often stimulates heated debates. My closing argument is invariably the same. Pretty code is most often the preference of those that get paid to program. Those of us that program for profit tend to lean the other way. The matter of style, inside a program, is a personal decision.

Programming may involve artistic talents but a program itself is seldom seen in an art gallery. With deliberate effort we can develop a style of coding that is efficient and easy, based on an awareness of how the interpreter itself works. Adopting habits encouraged by "experts" in other languages, and those with no practical experience in any programming language makes no sense in a profession that depends on our ability to reason logically.

At given moments in time we all vary our inclination, left or right, depending on the pressures of a particular programming problem. With a little luck we will have been found to have leaned in the right direction in a majority of cases.

Hopefully, some of this will enhance your inclinations. What has been displayed here was not learned in school. It has been my style, for quite awhile, reflected in a lot of programs written in interpreted BASIC. Programs that have paid for a lot of biscuits, even though some would not likely score a passing grade in a classroom.

Chapter 11 = DESIGN

What a program should look like is covered in Chapter 10. How to build programs quickly and easily is in Chapter 12. Chapter 13 suggests some coding techniques for implementing various aspects of a design. In this chapter are some ideas on how to first decide what a program should contain, where, why, and even, how many programs are needed to satisfy the requirements of a complete application.

The four subject classifications--style, design, method, and technique--are somewhat arbitrary, but chosen to minimize redundancy, and to permit concentration on one theme at a time. None of these can stand alone, however. They are all tightly interrelated. It is the sum of these ideas that make it possible to generate custom application programs cheaply, in BASIC, that work efficiently and safely.

Efficiency, as a design subject, has at least seven parts:

+ Run time performance of a given program.

+ The time involved in switching between programs in a total application set.

+ Usage task-times, i.e., the proportional amount of time spent doing things like file maintenance, posting business transactions, printing reports, changing printer forms and making back-up copies of software and data files.

+ The manhours involved in building and debugging programs.

+ The probability of costs for diagnosing failures and making fixes.

+ The labor-risk for making future modifications because of changing requirements.

+ Operator training and documentation updates.

Safety, as a matter of design, has at least four parts:

+ Hardware failures; namely disks, disk drives, and all kinds of electronic breakdowns.

+ Programming bugs; yours, the interpreter's, DOS, and its hodgepodge of parts.

+ Corruption caused by "foreign programs".

+ Operator foul-ups of all kinds (also known as "OFU").

Nearly all of this book to this point has been concerned with obtaining maximum performance for a given program while it is executing. A fundamental theme has been: The fastest running programs are those written with a full awareness of how the GW-BASIC interpreter works.

Most full-scale applications are made up of several programs. Getting from one to another, efficiently, is often a critical function of design. Poor design choices here can sometimes cripple programs that would otherwise run like racehorses.

All efforts for attaining efficiency, in both the man and the machine sense, can be obviated entirely if a hasty design does not fully consider safety issues. Mangled disk files will make any user unhappy. When it happens, they will be in no mood to compensate you for the time required to diagnose the cause of a failure, even when you can "prove" that the fault was theirs, an act of God, Mother Nature, a chip maker, or some other programmer.

Although some of what follows could be applicable when writing programs in almost any language, the emphasis here is on that which is peculiar to GW-BASIC, specifically. Little emphasis is needed about particular types of applications. Payrolls, games, and toy programs all deserve some thought about design. Only the amount of time should differ, not the quality of what goes into such intellectual exercises.

Nearly all programs have to access disk-based files of one type or another, sooner or later. To do so quickly, and safely, consider the ideas listed here, and the rationale on which they are based. At design time. In that interval of deliberation before coding is begun. Coding errors can cost a little time to find and fix. Design errors on the other hand, can bankrupt computers, others, and you.

Whole stacks of books can be found in the libraries regarding

software development projects. They invariably propose writing a lengthy functional requirements document, a project plan, and an installation and testing plan. Different authors choose to communicate their ideas in various terms. Some break the whole into different pieces. Having studied many such books, and having practiced their advice to one degree or another over the years, with gangs of coding coolies, and in lone-star efforts, here is my synthesis of it all, in this environment.

Build menus first. This does two things. Menus are a "list" of what functions an operator must be able to do. They also provide an "outline" of the major functional tasks that must be programmed for. Hence, this work approximates what would go into a requirements document, but it is also an end-use product, not just reams of paper.

Build data entry and display masks next. Concentrate here on record layouts, field sizes, and the like. Thought has to be given at this point for reporting requirements so that what will be needed eventually, will have been captured somewhere. This effort also relates to a "requirements definition", but it too results in end-product usable output.

Build skeleton programs next. These are really dummies; the only one that has a modicum of intelligence is the one that provides for menu selections. The others merely display the various screens that will be seen by an operator, and give an impression of how "friendly" the final product will be.

Build a few help screens next if on-line help is a basic requirement. (If not, resort to an abbreviated piece of written documentation.) The subject emphasis here is on how installation will be done, and the interrelationship of this application to others in the same system domain.

The sum of all efforts to this point is akin to what is alluded to in many textbooks as a "prototype design". You now have a demonstration tool for interacting with the intended user to ascertain that all of the bases have been covered. And you have a "plan" that outlines what "modules" will be needed. (It is from this that estimates regarding schedules and costs can be made, also.)

Now see how the above suggestions relate to BASIC: The menus, entry masks, and help screens are built and used with BSAVE and BLOAD. Chapter 6 contains some technical specifics about this;

Chapter 12 suggests some ways to do it easily; here we consider some pertinent design issues.

The use of LPRINT can cause problems sometimes in programs that also do BSAVE and BLOAD. The obvious solution is to OPEN the printer as a device and PRINT to that "file". Read this another way: Forget the word LPRINT.

The disk space needed for BSAVE/BLOAD text files is double what would be needed normally, e.g., 2-bytes per character rather than one.

BSAVE (and BLOAD) can only address a DOS file name. And always, an entire file. In monoadapter machines this means every screen, or partial screen, is a file. Multiple "pages" can be saved and loaded as a single file with color adapters, but BLOAD always loads an entire file (all pages that were saved in one file). A lot of screens can mean a lot of file names, which can have a negative impact on the time required for directory searches for all file names, including those for programs and data files.

CGA machines "flicker" when you do a BLOAD into video RAM areas. While this is not a tremendous problem, it is a fact of life (and difficult to explain to an operator, even though it is often only a minor annoyance.)

BSAVE to a flex disk is very slow; BLOAD is a little faster, but the screen appears to be painted in "chunks". (A floppy sector is 512 bytes; DMA transfers are done one sector at a time.)

For systems that have only floppies, if the operator removes the disk with BLOAD screens on it, PRINT will have to be used as an alternative means of communication in error situations.

BSAVE and BLOAD work from point-A to point-B as continuous and contiguous strings of bytes. In practical terms, partial screens can only be done as consecutive, full-width lines; pop-ups and pull-downs must be designed as screen-wide blocks.

As with all design issues, there are trade-offs, of course. A significant advantage to the use of BSAVE and BLOAD screens is that programs themselves are much smaller. And much faster.

Because character attributes are saved (and restored) along

with text, and lines and boxes, there is far less need for a lot of nitty gritty lines that do LOCATE, COLOR and PRINT. And the text itself is outside of the program, freeing-up even more space better used for "intelligent procedures" and working storage areas.

BSAVE screens can also be used to cheaply pass along information to other programs. Suppose an operator selects a report option from a menu, then the menu program chains to a general purpose report writing program. Just before the chain takes place, mark the operator's selection on the menu and BSAVE it. When the report program starts running, it can examine the menu to see what it is supposed to do. (The use of COMMON and other inter-program communication techniques is explored later, but remember this example of a simple alternative.)

Now it can be further seen why this chapter on design began with a focus on using BSAVE and BLOAD screens: A decision in this regard can be significant in how an application is mapped overall, especially in terms of how many individual programs will be needed, and what they will each do. Typically, because of the use of "screen files", more intelligence can be built into a single program than would otherwise be the case.

In GW-BASIC we have to always keep in mind that we have to live and work within a 64kb bucket. Oddly enough, this is usually ample; seldom does it have much influence on how an application is best divided into component programs, assuming you agree with the following attitudes:

+ For performance reasons, short variable names are used, and all variables are reused, to the maximum extent possible.

+ For both performance and labor reasons, dense code is best.

+ For labor and space-saving reasons, screen files are easy and economical.

+ Once the interpreter is loaded, the big drag is over. An end-use program can be RUN-loaded or CHAIN-loaded pretty fast if individual tokenized program files themselves are fairly small.

+ Program files will naturally tend to be small if their bulk is mostly procedural code and not a lot of "text".

Assuming concurrence with this background, how many programs, and what each will do can align with what the user wants to do. A simple example can be used to convey this idea; of all of the different types of business data processing applications, the simplest one that comes to mind is a stand-alone General Ledger accounting application. The basic functional requirements for this one comprises a small list.

Master file: Small records, one per account number.

File maintenance: Operator needs to be able to create new accounts, delete obsolete ones, and make minor corrections to static descriptions and the like.

Transaction file: Each accounting entry is a "record".

Posting operations: Accounting entries are debit and credit adjustments to an account balance. The current balance is maintained in the master record; each posting line is tacked onto the tail end of a transaction file.

Reports: Two kinds, essentially. Journals are listings of records in the transaction file. Profit and Loss Statements, Trial Balances, and the like, come from the master file.

Ignoring mechanical details for the moment, see how this simple list can be expanded to embrace almost any data processing problem. The others are simply more of the same, no matter how sophisticated their overall requirements are. Because of this view, a general-purpose design template can be described.

Master file maintenance: One program for each file. It will OPEN its master for both reading and writing. For ISAM file situations (q.v. Chapter 13) two files may be involved--the real data file, and its associated index. In sophisticated applications, master-file maintenance programs may also have to OPEN ancillary files in a read-only mode for validating operator entries, or for merely providing visual references.

Posting program: Only one (usually). In concept, this is a data entry program. It is also the guts of an application. This is where it all happens in terms of what a "program" can do. At various points in this process most all master files must be open; some for reading and writing, some for read-only. Output is to at least one file; sometimes to more than one. The output from posting invoices, for example, may be an intermediate file to be used to drive another program

that prints forms and updates accounts receivables balances.

Report program: Maybe one, maybe many. A distinction must be made between those that only read-and-list, and those that list information and do concurrent updating of master file records on the fly (or at the end of the report run).

While the above is not a radical view in any language, on any computer, it is especially apropos to BASIC and DOS. The three types of programs--file maintenance, posting and reports--are a natural definition of separate programs for at least three reasons: Performance, space, and safety.

Performance: The need for speed is different for the three major types of tasks that an operator does. File maintenance and posting are both operator paced, but differently so in the sense of key-thumping burst rates and entry rhythm. Reports, on the other hand, run largely unattended once initiated.

While doing file maintenance, little paper shuffling goes on. With the exception of creating new records, not much time is spent on any one record. Emphasis here should be on fast record access, and rapid displaying of entire records on the screen.

During posting an operator's eyes are focused on input source documents far more than they are on the screen. Although a lot of disk accesses may be needed for each transaction line, little of that data is actually displayed. Because of an operator's natural inclination to see-a-line and post-a-line, it is best to echo each columnar entry quickly, and do all cross-file validation checking at the end of each line.

Reports are typically printer-speed paced. This is still often true today in business operations, notwithstanding the popularity of large capacity "buffers", and the sometimes supposed advantages for things like spooling and so called "background multitasking". For many "reports" like invoices, monthly statements, and paychecks, further data processing cannot be done until those runs are fully and successfully completed, and backed-up, no matter how it is done. Design emphasis here must center on efficient coding of printer output sequences.

Space: The mapping of an application into separate programs on the "natural basis" suggested here often causes the best

fit automatically in terms of the 64kb maximum space limit per program.

File maintenance programs, for example, need to fully define all record fields. Large chunks of procedural code are often needed to let the operator move around in a record, and for thoroughly checking all keyboard input. This program is also the one that must do full index maintenance for ISAM files. Nearly all of these functions often require large sequences of rote code that are unneeded in other programs within the same application set.

A bookkeeper sees a debit and credit as two different things. Programmers see them as essentially the same; only the signs are different. With small effort, a single set of procedures can accept input for either an invoice or a credit memo. Two different entry masks, with different "headings" will benefit the operator. By designing them so that columnar alignment is the same for both (or nearly so), only a single set of subroutines are needed internally.

No matter how it is done, a program that prints a report is mostly rote procedures, but seldom very lengthy. It is here however, that memory constraints can sometimes impact the design of arrays for accumulating totals and intermediate subtotals. In some, large "lists" must be constructed and sorted in order to provide output in a required presentation sequence.

Safety: Updating disk files is a risky business. Updating existing relative file records is the least risky--only that particular file's contents is in jeopardy at any given moment. Adding new records to a relative file is akin to sequential file output as far as DOS disk-mechanics are concerned. And so is creating any kind of new file. If a failure occurs while DOS is fiddling with the FAT, an entire disk may become totally useless in a pragmatic sense. The onus is on us to deal with these risks. Buck passing stops here; we have to cover fully for the sins of DOS, clumsy operators, and all "cheap" programs allowed to run in our machine.

Because master files are relative files, invariably, file maintenance programs must contend with both record updating risks, and those inherent to file-stretching operations.

Posting programs do not normally have to stretch master files,

but, they do have to update existing records and continuously add-on to the end of transaction files, be they of either type, relative or sequential.

Report programs present the least systems integrity risks if they are of the read-and-list variety. A failure while reading records is unlikely to harm anything on a disk, save for the usual risks of physical damage caused by electronic or mechanical breakdowns of the drives themselves. When it is necessary to store report accumulations in temporary work files, or do batch updating of data records, that program must contend with file and systems integrity issues with the same sense of responsibility as any type of data processing program.

Having now decided how many user task-oriented programs will be needed, and what each must be responsible for, it is time to decide how to hook them all together.

As a matter of habit, my preference tends to be to do all menu selections from within a single program. For several reasons. To you, and an operator, a menu program can be seen as a table of contents. But from slightly different perspectives.

From a menu an operator can choose what type of work they want to do next. When finished with an operation they can return to the menu for another task-oriented selection. Menus provide a condensed definition of what an application can do, and, serve as an "office manager" for enforcing procedural discipline.

The menu program can also serve as an "application manager". It is the logical place to do global housekeeping chores so that task-oriented programs need not have to contend with the mundane. Having solicited today's posting date, for example, the menu program can simply pass it on to other programs, so they can assume they will receive only valid data. This is also an obvious place to do nuisance chores like converting dates to Julian, if that format is needed by several other programs as a static value.

Another type of "global chore" suitable for a menu/gateway program is usage logging. An operator's menu selection can be "logged" before the called-for program is loaded. By causing all programs to exit to this common program, success or failure status indicators and activity counts can be passed back to a single routine responsible for maintaining a log of who did

what to whom, and when--an indispensable aid for doing fault isolation and diagnostic work.

Another reason for favoring a single gateway concept for an application's basic architecture is that it is a good place to take care of configuration idiosyncrasies. Once this program determines how many disk drives are available, for instance, a mere status indicator can be passed to other programs so that they do not have to duplicate unnecessary procedural tedium.

A similar but often more laborious task can be accomplished once, in this one program. Modern printers have memories of their own. To ensure that a printer has not been left in an unpredictable state by an interim process, the menu/gateway program can arbitrarily reinitialize all printer options just before a call to any program that makes use of that device.

The above not only precludes redundancy in all programs that print, it also localizes into one area all of those eight ball codes that are likely to be different for various printers. The obvious payback here comes when a user opts to upgrade his printer, i.e., only one program will have to be "updated".

Now for the clincher: Application and system integrity. When an application is first started, the "menu program" should do a thorough check that all is Ok before any type of processing is allowed. (If not, a RESTORE should be enforced.) Assuming all of the programs within the set are well behaved, subsequent (but simpler) checks can be made on each return to this single program to ensure that nothing went haywire while its back was turned.

By making the start-up gateway process responsible for making sure that all programs, screen files, and data files are in fact resident and ready, discrete programs can be written so as to make a number of assumptions without having to contend with every conceivable eventuality.

Obviously much of the foregoing would be an equally valid set of considerations no matter what programming language is used. Before we begin slapping up code in GW-BASIC, however, there are several perversities that must be reckoned with as design-level issues.

Although the manuals devote but a few sentences to describing how RUN and CHAIN work, there are tremendous differences to

consider in choosing one vs. the other. Those factors will be enumerated shortly, but there are fundamental problems to consider about how to get back and forth smoothly from DOS to BASIC in the first place.

There are five sizing assumptions built into the interpreter. Those default values may be altered by the use of "switches", when GWBASIC.EXE is loaded. Whatever those values are--the default values, or your own--they remain constant for all programs that run from then on. The only way to alter these values is to reload GWBASIC.EXE; the only way to do that is to exit all the way back to DOS command-mode level. And that takes time, especially for the reload of the interpreter.

Suppose at least one master file in an application has records of 512 bytes. Suppose a posting program has to open six files concurrently. We might launch the application with a command line similar to this:

```
GWBASIC AR-MENU /S:512 /F:9
```

My latest manual says each F-number costs 194 bytes, plus the size of the S-number, for each one. I have not yet figured out how to make accurate use of that advice. Using GWBASIC.EXE version 3.23, the difference reported by FRE(0) in this case is 5772 bytes, vs. what it would be for a "default load". (54528 vs. 60300, with no program in memory.) It would appear from this that each file costs 642 bytes. If we increase the number of files to 10, the amount of memory reported is 760 bytes less than for /F:9, but 9 times 760 does not equate to 5772. On the other hand, 512 (my file size) less 128 (default size) is 632. And 632 times 9 is 5688, which is pretty close to 5772, really.

> Note: The manuals also say that the S- and F-switches are ignored if we do not also specify an I-switch. Ignore that tidbit. To the best of my recollection, slash-I itself has been ignored by all versions of the interpreter since about 1982, or so. Slash-I has no influence on anything at all, today, as far as I can see.

When doing design estimates my rule-of-thumb is:

```
R = Maximum record size, plus 130
N = Maximum OPEN-data-files requirement, plus 3
```

then R times N is the approximate constant memory cost for all programs that run after the interpreter is once loaded. (The

plus-3 for N, by the way, is for the keyboard, monitor, and printer; even if we do not plan to open them explicitly, the interpreter itself implicitly anticipates need for them, as mentioned in Chapter 6.)

At the risk of kicking a bent bucket: All of the sizing options specified when the interpreter is loaded are fixed overhead costs for all programs, whether we opt to use those built-in, or, explicitly nail them down ourselves.

Now given that we know how many programs are involved, and how big our run-time bucket is, we can decide on a design strategy for getting from one program to another. In BASIC there are basically two choices: RUN and CHAIN.

Of these two, RUN is nearly always a best first-cut choice. At a conceptual level, CHAIN has but one advantage: "Information" can be passed from one program to another (via COMMON, or CHAIN with ALL). If we opt to use RUN, we must use alternatives for passing "application parameters" from one program to another. Although it may take a little time to store such items with one program, and retrieve them in another, that time may actually be less than would be the consequence of using CHAIN to pass "common variables" to a called program.

> Common? My heritage is, "... the common folk". That is, without refinement in language or manners. Some wag chose COMMON as a word denoting data (in variables, in memory) that belongs to a community of programs. The first, most simplistic definition of this word, according to Webster, would seem to make it a good programming language key word.
>
> Dictionaries and BASIC manuals use about the same number of words to define COMMON; in the latter case it is not nearly enough, because, there is so much that is not commonly known. In fact, COMMON, in BASIC, also lacks refinement in language, and its manners are often deplorable.

Chapter 3 describes how variables are stored, and searched for, by the GW-BASIC interpreter. It also made passing reference to the potential performance impact of using CHAIN and COMMON: Variables passed to a second program will be at the top of the stack in the chained-to program. Variables declared in that program (that are "uncommon") will be slower to reference; a chained-to program may run slower as a consequence. A long

list of COMMON variables (or the use of ALL) deserves some long thought if optimum performance is desired.

The order in which COMMON names are named has no significance. Nor does it matter if they are named in one or several COMMON statements. Nor if several such statements are placed one after another, or scattered throughout the declaring program. Although COMMON statements do not have to be "executed", they cannot, in fact, be conditionally executed.

When CHAIN is executed the interpreter scans the entire program then in memory and looks for COMMON declarations. It makes no difference where they are located, unless they are on a line following THEN or ELSE, in which case they will be ignored altogether.

It does not really matter how many different COMMON statements there are, or if some of the variables they name are duplicated in different places, or even twice within a single declaration. Sometimes. Read on.

In a simple context, here is what the interpreter does when it hits CHAIN:

It scans the program looking for COMMON statements (unless CHAIN specifies ALL).

COMMON statements will be recognized as such in any line that could be "executed", even if in fact, program flow never hits that line, or gets that far.

If COMMON is inside a FOR/NEXT or WHILE/WEND block it will be "processed" when it comes time to CHAIN, even if such blocks of procedures are bypassed during execution because their "conditions" are never met.

If COMMON is contained in any IF statement, that declaration will be totally ignored. (Although the conditional logic itself will otherwise seem to work just fine.)

As each variable is encountered in a (recognizable) COMMON statement, that name is located in the variables storage area, and tagged. If a name was declared to be COMMON, but never actually used anywhere, it too is effectively ignored; nothing will be passed along, not even the name.

When the scan for COMMON statements is finished, all of the

variables that were tagged are "compressed" into a block, still stacked in the same order as they were declared while this program was running.

If a variable is effectively empty--a zero, or a null string--the name will be kept in the block of variables to be moved, notwithstanding there is no "data" to be conveyed.

If a variable points to a string literal, that text is copied at this time, from up inside the program statement, down into string space. Grrr. This can be a beast. (Re: Strings in Chapter 4.)

Variables named in FIELD statements--that are also named in COMMON--are also preserved in the block of variables to be passed on. At that point, the address pointers for those names are aimed into record buffer areas, as they still will be after the CHAIN takes place, notwithstanding whether their associated file has been closed, or left open. Carefull....

Both simple variables and arrays may be declared as COMMON. Both "tables" are passed as two distinct blocks of bytes; the simple variables are on top, followed by the arrays. Recall that arrays are passed en masse; it is not possible to pass subsets of an array.

If CHAIN with ALL is specified, no preliminary scan is done; the presence or absence of COMMON anywhere is effectively ignored.

The above description of how COMMON works is merely an enhanced version of what can be gleaned from the manuals (save for that about COMMON not working with IF, which can only be learned the hard way). Nowhere is their enough detail provided to enable us to make design decisions. Much more must be known. A poor choice at this point may not become evident until a lot of coding has been done. To reverse our strategy that late in the game can cost a lot of hours of hard work.

IF variables named in FIELD statements are carried forward via COMMON, their data will be passed along also. This is true even if you CLOSE a file before doing the CHAIN. (CHAIN does not close files; they will still be OPEN in the chained-to program unless you explicitly CLOSE them.) If you do CLOSE a file then CHAIN, the data pointed to by all FIELD declarations is still that contained in buffers associated with their file number. In

the called program, if that file number is reused for the same or a different file, the data associated with the original FIELD statements are effectively lost.

And remember from Chapter 3 that arrays are stacked in working storage after all simple variables--the naming of new simple variables in a chained-to program will cause a slight delay while the interpreter shifts the arrays downward to allow for insertion of newly named simple variables in the upper table. And it does it repeatedly. If the incoming program names ten new variables, the arrays get moved ten times. If several arrays are involved, and especially if they are large, the "delay" can be a lot more than "slight".

The manual says a given variable cannot be named in more than one COMMON statement. The interpreter does not check to see if we make a "mistake" in this regard, however. You can violate this "rule" and get away with it. Seemingly. In the case of simple variables, there is no noticeable penalty for such infractions. In the case of duplicate declarations of arrays, the penalty is akin to capital punishment.

If an array is mistakenly named twice in one, or in different COMMON statements, things really slow down, although it may not be noticed until the chained-to program begins running. But, run it does not. In fact, it will not even walk very fast. It will barely crawl along, actually.

The lethargy contracted by making a duplicate declaration of an array to be COMMON is amazing. Until you try it, you would not believe your computer could run so slow. Even LIST will merely dribble across the screen. Worse yet, there is no known way to overcome this malady, save for canceling the whole show. Once infected, the interpreter will thenceforth limp along, no matter what you do short of going all the way back to DOS and reloading GWBASIC.EXE.

The above malady can be avoided if we are careful and play by the rules, of course. Unfortunately, the rules are not very well defined. Nor enforced.

There is only one tidbit of advice in the manuals about COMMON statements: "... it is recommended that they appear at the beginning." Rationale for that little gem is not intuitively obvious. (Recall that the entire program must be scanned to find all COMMON statements, be they up front or scattered hither and

yon.) We would be far better advised if they had used this space to tell us the real consequence of opting to CHAIN with COMMON, or, CHAIN with ALL:

Garbage clean-up is done of free-string space, even though it may not be needed, really, before CHAIN is actually executed. Chapter 4 describes garbage, and the costs for it, and how to avoid it. There is no avoiding an implicit FRE("") however, for CHAIN with COMMON, or, with ALL. Trap this bee in your bonnet lest you be stung later by a decision to use CHAIN in lieu of RUN as a design strategy.

At a glance, CHAIN may seem to be an easy way to accomplish interprogram communication of global information. Sometimes it is practical. Perhaps. From one program, to one more, on a one-time basis. As a rule of thumb, that is my advice. For any application that is designed as a set of programs that must be called recurrently, RUN is a far better choice than CHAIN. No garbage clean up is needed, or done, on a RUN.

Because no garbage clean up is done on a RUN, it lends itself to very good use for more than simply calling another program into memory. A simple RUN statement followed by no arguments is an effective way to restart a program already in memory. Obviously. It is an effective way to clean up the garbage in string space without suffering the cost of a FRE(""). A trade-off has to be made, however, for the time needed to "save" what may have already been "computed" before a re-RUN can be done. Of course. (Which is effectively no different than what must be done in any case of using RUN to switch programs, be it the one already resident, or not.)

A more subtle design use can be made of RUN to a line number: Multiple programs bound together as a single load module. This is often a handy trick for achieving maximum performance.

+ A given program should use the fewest variables possible.

+ Variables should be declared in prioritized order, based on which ones are used the most.

+ Once a prioritized stack of variables has been built, there is no practical way to re-order that stack.

+ Many variables needed in one phase of a program's operation may not be needed later on. And vice versa.

If we design a given program as really two, and go from phase-1 via a RUN aimed at the first line of phase-2, we can optimize both phases to their own best ends. By mapping the phase-1 and phase-2 blocks of code so that they are physically one program, they can both be brought into memory with a one-time RUN. (Also, obviously, subroutines can also be shared by multiple phases, thus saving space by minimizing redundant coding.) Thus, RUN to a line number can be used for achieving greater efficiency in several ways.

We come now to the issue of how to best do what is often called interprogram communication -- the fancier term for what COMMON was intended to be used for. A small preamble will provide the background for several suggestions.

Back about the middle of this century the concept of overlays became a popular concept in machine language programming. When there was not enough room in memory to hold all of a program, a small chunk of memory was set aside for transient portions of coding. If a needed segment was not already resident, it would be "overlaid" on top of whatever was already present in the overlay area.

> Note: As recent as 1980 or so, we often had to fit our BASIC programs into as little as 4kb of usable coding space. So, CHAIN with MERGE was invented. It is as clumsy today as it was back when it was first conceived. It is unfathomable to me that any professional programmer has ever made use of this gadget. So many superior alternatives exist, CHAIN/MERGE deserves no real contemplation as a designer's choice.

The subject of overlays has been introduced because that is in fact what all interpreted BASIC programs really are: The interpreter itself is a program; it pre-allocates a 64kb work area to contain address pointers, counters, I/O buffers, and, your program. When we RUN or CHAIN, the incoming program will overlay whatever (BASIC) program is already contained in that portion of the interpreter's working storage space set aside to hold our programs. A GW-BASIC program is a "data area overlay" managed, by the real program that is running, i.e., GWBASIC.EXE itself.

Given this perception of what a GW-BASIC program is, and by knowing where things are in memory, some obvious opportunities can be seen about where things can be put so that they may be

passed along to subsequent processes. Here is another short definition of how the interpreter's working-storage area is mapped. It is an abbreviated version of the five subdivisions described in Chapter 4.

Block-1: Interpreter's own parameters and your I/O buffers.

Block-2: Tokenized BASIC program.

Block-3: Variables storage area.

Block-4: String space (so called free space).

Block-5: Interpreter's run-time stack.

Bearing this simplified map in mind, remember that blocks-1, 2, and 5, are static in size while a program is running. Block-3 grows downward. The space in block-4 is used from the bottom up. An incoming BASIC program is an overlay; it does not wipe out anything in block-1 or block-5.

When we use CHAIN or RUN to cause a program to be brought into memory from disk, it goes into block-2. At that point, if we had used CHAIN to communicate some variables, they would be shifted into block-3 so as to end up immediately following the procedural code in block-2. In the event strings are passed along also, they would be compacted at the bottom of block-4.

In the event we use RUN in lieu of CHAIN, the incoming program still overlays into block-2, from the top downward, but the interpreter simply reinitializes its variables-pointer to the top of block-3, and its string-space pointer to the bottom of block-4.

Notice the similarity: What needs to be overlaid, is, and, the interpreter's pointers are adjusted accordingly. Areas that are not overlaid by incoming procedural code, or by any variables passed along, are not "erased". Logically, those bytes that are not overlaid are merely residue insofar as the interpreter itself is concerned. Thus, obviously, we can cause that residue to contain whatever we want, deliberately. When a subsequent program first begins, it can then sneak a quick PEEK at the litter left behind by a previous process.

This is a cheap trick for passing along a few parameters from one program to another. No doubt. This is neither the time or place to deliberate sacral issues. The foregoing is not meant

to offend anyone's sense of what constitutes "good programming practices". But, by golly, it does work. And it is an easy and efficient way to make use of RUN, and avoid the inefficiencies of an ill-bred CHAIN.

One of the more meager things that we sometimes need to do, when we RUN to another program, is to name the line number that execution should begin on. Hard-coding that line number in the calling program is the last thing we want to do, obviously. (If we renumber the lines in called programs, the callers may have to be modified as well--a real pain in the pocket). Textbooks and teachers would have us pass a parameter in a variable to be examined by called programs so they can GOTO or RUN-to an appropriate line number known unto themselves.

Knowing what the interpreter still knows, that which it already knew after another program is loaded, however, we can make use of its own memory without creating artificial intelligence of our own. The following suggestion is a simple example, but slightly less risque than the one previously offered. Yet, it too depends on an awareness of an "undocumented feature" of GW-BASIC.

ERR and ERL are two parameters that the interpreter keeps track of in its own working storage area (up in block-1). ERL will always be set to zero by any form of RESUME, and by a RUN, or a CHAIN. ERR, on the other hand, is never actually reset; it is merely updated when an error occurs. Meanwhile, it contains the code-number of the last error that did occur, however long ago that was. And it still will, after a RUN or CHAIN takes place, although, this is not mentioned in the manuals.

See the possibility: Do an ERROR 101, or ERROR 102, then a RUN, for example. The incoming program can then look at ERR to decide where it should GOTO on the basis of the code in ERR. Another quick, cheap, and not particularly risky trick, even if it is not a particularly handsome one.

Oftentimes only a very small amount of information needs to be passed from one program to another. Like, for instance, a "switch", or a menu selection indicator. Here is another trick, especially attractive in such cases. Sometimes. Use a few bytes in the system's Interrupt Vector Table.

Low order memory in DOS machines--beginning at address 0:0--contain (mostly) segment and start addresses for routines that

are accessed via BIOS and DOS functions: "interrupt vectors".

This table is organized as word pairs, two bytes per word, thus each interrupt number is a 4-byte offset from zero. By reason of "PC compatibility standards", certain ranges of interrupt numbers are reserved for special uses. The software interrupt numbers ranging from 20h-3Fh, for example, are reserved for DOS. Interrupts 60h-67h and F1h-FFh are "reserved" for "user programs". In theory then, we can store addresses or codes, or anything we want to in those bytes pre-allocated for "us users".

To pass a menu-letter then, for example, do a DEF SEG=0, and POKE 384,ASC(Q$), assuming Q$ contains the letter we want to pass to another program. And 384 is 4 times 96; 96 in decimal equates to interrupt 60h; multiplying by 4 gives the physical offset into the interrupt vector table for that interrupt.

In a follow-on program, DEF SEG to zero and PEEK(384) to get the value stored by the previous program. Yea! Very fast, and very easy. However: You are a "user", but so is anyone else not on the payrolls of the folks that write BIOS and DOS software. Like the people that supply TSR--Terminate and Stay Resident--software. Like pop-up calendars and clocks and notepads, and the like. They too can use these addresses reserved for all of us "users". Caveat.

Purists will tend to like purer forms of interfacing: Store global application information in interprogram communication files, data files specially contrived for just that purpose.

This is obviously a clean way to do things, especially when what needs to be communicated must survive the big gap: When it is necessary to quit BASIC and return to DOS, even if only for a short interlude. Which brings to mind also, one of the most awkward design problems that has to be contended with in the dizzy world of DOS and BASIC: Intertask communications with DOS itself, or other "packages" written in any language.

One very serious consideration that must be made at design time is avoiding unnecessary disk fragmentation. Special files used for interprogram communication are typically pretty small, but highly dynamic. If we manufacture them on the fly, and delete them frequently, while also creating and stretching real data files, we will also compound that old DOS Nemesis, disk space fragmentation. Before we opt to devise "extra overhead files" we should consider if some of those needed for other reasons

can do double duty as interprogram communication vehicles.

No intellectual feats are needed to see some obvious choices, but here is a short check list to keep from overlooking the more mundane.

DOS BAT-files themselves may be "read" by a BASIC program to obtain system configuration data. REM lines in BAT files can be deliberately devised, easily, for just that purpose.

DOS BAT-files may be modified by a BASIC program (or even, manufactured from scratch). A clean trick for interfacing to custom-created batch files is to put a "stub" in the one that starts the BASIC ball rolling in the first place. A stub of this type may merely be the name of a BAT-file that may or may not already exist. Obviously, a generic name-stub can be used as an alias for a variety of batch procedures that can be generated on an as needed basis by any BASIC program.

A simple variant of this second idea can be to pre-code a set of batch files to cover various situations. Then, in BASIC, merely rename the one needed at a point to have the same name as that already implanted as a stub-alias in the BAT-file that controls the whole show.

To preclude a BASIC program from attempting something that is not obviously impossible, the prevailing CONFIG.SYS file can be examined. Obviously. In a few far-out cases it may even be reasonable to custom manufacture a CONFIG.SYS on the fly, and artificially stimulate an automatic reboot.

The design thrust of the above ideas center on avoiding having to create separate, "special files" for the exclusive purpose of achieving interprogram communication. There is yet another file that can be read, and modified, for pre-conditioning how a program runs: Program files themselves.

This is an especially effective way to automatically "install" an application: An installation program can be written to OPEN each of the programs in a set and implant "option switches" as needed. The obvious advantage of this trick is that those programs will run smoothly from then on, without having to do any external research every time they are run. (Chapter 13 has some specifics for doing this; here we are simply looking at various design choices before coding commences.)

While some of the above ideas may offend the sensibilities of

those dead set against programs that modify themselves, such tricks may often be far less risky than "conventional coding" alternatives. Like making use of DOS environment strings, for example, or using systems service calls, or PEEK, to see what DOS knows, or thinks it does, about what is going on outside your BASIC box.

Only a small counter-argument is offered here for those with platonic ideals: Who do you trust the most? Those who have blind faith in DOS, or rely on undeserved faith in BASIC (or any language product) may not be blind, but chances are, they have not had to suffer many "upgrades" of successive releases of either.

There is probably nothing more exasperating than having to modify old programs that no longer work just because something was changed in a recent release of systems software. Even when such changes may prove eventually to have been a good idea.

When we design programs so as to be dependent upon our own devices, we are merely susceptible to our personal genius or ineptitude. In either case, faith in ourselves should prove more worthy of our trust in the long run. The more we know about how DOS and BASIC work, and the longer we use them, the less we are apt to trust either of them at all.

In all events, we are ultimately responsible for not only our end product application, but for the continued integrity of a user's system: A responsibility that must be kept in mind at all times. When confronted head on at design time, thoroughly, the burden becomes less onerous while writing the programs that will actually get a job done.

While some of what has been suggested here is not encouraged in many schools, distinction must be made between getting good grades, and getting paid. Uncouth compromises are frequently a real life fact in commercial endeavors.

Those last thoughts bring to mind one more design consideration that can cause us to rethink the whole problem. We may need to take a slow look at QuickBASIC before we commence coding:

The 64kb memory paging considerations for BASIC programs has dramatically different, but larger boundaries.

It is virtually impossible to predict with any accuracy how

large a given compiled program will ultimately be.

Because QuickBASIC is an in-memory compiler, it is nearly impossible to anticipate when you will run out of memory. There is a risk with large programs that a maintenance update in the future will necessitate completely revamping some or all boundaries between programs and subprograms.

COMMON rules are so different it is nearly impossible to use it in GW-BASIC programs that are to be compiled some day.

CHAIN differs only slightly from RUN--the pragmatic choice is to design for RUN, and forget the word CHAIN.

Compiled programs are complete programs; when we RUN or CHAIN from one to another, an incoming program completely replaces its caller.

The only practical choice for interprogram communication is via disk files; unconventional tricks used in a GW-BASIC program simply will not work in compiled programs.

Having digressed slightly, and perhaps provoked some rethinking about overall design strategies, a final attitude is offered on the themes of trust, faith, and ethical compromises. When we use GW-BASIC we may be compelled to use some nefarious tricks; when we use QuickBASIC we are vulnerable to tricks they play on us. Because they change so many things so often, with no hint of what is coming next, if we are designing for the benefit of others, GW-BASIC should be an automatic choice.

In the final analysis, it is better to trust what we know than to take chances on the unpredictable. What has been listed in this chapter are the major thoughts that flit through my mind when designing new applications. Whether you agree with what my experience tells me should be considered, few can argue that serious premeditation is needed before new programs are born. Hopefully my list provides some food for thought; it is a diet that has enabled me to raise a lot of healthy programs.

Chapter 12 = METHODS

The order of presentation of subjects in this book is backwards compared to that of most other texts. On purpose.

The "user's guide" portion of manuals is always up front. It tells you how to SAVE and LOAD and EDIT. The reference section of manuals is arranged like a dictionary; it assumes you know how to program, but occasionally need to research the syntax or semantics rules for specific key words.

Textbooks that teach how to program usually begin with problem analysis and proceed with how to organize work tasks. Rarely do they broach the subject of what is efficient in the context of how the internals of a language actually works, nor to the extent that such considerations should influence our decisions about choosing one technique vs. another.

The order of presentation of subjects in this book was chosen on the assumption that you already know how to SAVE and LOAD and EDIT. And that you already know most of the key words of GW-BASIC and how to use them. And that you already know what an application's functional requirements are, and can perceive how those problems might be solved with a computer.

Given these assumptions, this book began with a look at how the interpreter works, with clarification of what does work as the manuals say, what does not, and why some things are more or less efficient in the mechanical sense.

The reason my essay describes BASIC from the inside out has but a single motivation: Efficiency. We are bound to write better running programs if we know how the interpreter translates a program. Hence the arrangement of Chapters 2 through 8. And Chapter 9 deals with strange encounters that would have been too far afield from the gristle presented elsewhere.

All of the preceding is fundamental rationale for suggestions about programming style in Chapter 10. That, plus my ideas in Chapter 11 on design are culled from experience in producing applications that do run efficiently. This chapter and those that follow are about how to do programming chores efficiently.

This seemingly backward presentation is logical on reflection. Turned around, we might achieve personal efficiency, but wind up with inefficient programs. Being driven by an awareness of what

produces the best performance, we can develop methods that are also efficient, without adversely affecting the quality of the end product. In fact, much of what has been suggested before lends itself toward this second objective: Getting the most, and the best, with the least effort.

Chapter 1 contends that it is possible to generate more useful code, cheaply, in BASIC, than in nearly any other programming language. Three arguments are fundamental to that contention:

+ GW-BASIC is the oldest high level language in the PC world; primitive algorithms written even ten years ago still work today, usually just as is.

+ It is likely today's programs will still work for a few more years, i.e., as long as GW-BASIC gets moved forward onto next generation hardware.

+ New programs can be assembled quickly by merging together pieces taken from old programs.

One probable reason for the long-term stability of GW-BASIC is because of neglect. The multitudes seem to want the latest and greatest whatever; interpreted BASIC has become an orphan by default in the fast moving world of PC gadgetry.

It would be inappropriate to enumerate the merits of other languages here. Little counterargument can be offered that several require far fewer lines of code to achieve a desired end (for some types of programs). Most have superior editors, as well. Some also have impressive lists of tools. Few lend themselves easily to the idea of reusing fragments of existing programs to create new ones, however. And, many of the newer languages are so young that there cannot yet be very many "old" programs in anyone's personal library.

The very fact that there has been little clamor for GW-BASIC enhancements actually works to our advantage. The more we use it, the better we are able to use it. For those of us whose income is directly related to our output, we make more by using what we already know: Time spent learning what's new can only be offset by improved productivity at some future date. In the PC industry, what's new happens so often that the future remains forever elusive. By the time we master what is new today, much of that experience will be obsolete.

There are really three ways in which we can take advantage of the fact that GW-BASIC has not changed much, or often. Not only can we maximize reuse of old code, our methods can become ingrained habits. Once again, time spent learning how to do old things a new way seldom produces immediately realizable profits. The mere fact that we must forget how something used to work is proof positive that what we did learn once must now be seen has having been, really, a waste of our time.

The third advantage to working with a "mature language" is that our own-code tools seldom need updating. By comparison, time spent developing a source code generator, for example, for some languages, may never actually be compensated because of having to constantly revamp such tools to stay abreast of changes in the language itself. Every hour spent writing or overhauling tools must be recouped twice, through improved productivity, to justify having fooled with the tool at all. For languages that are subject to frequent changes, it is risky to take time away from paying projects to develop new tools.

It is with these thoughts in mind that the following ideas are offered. None are revolutionary, nor are many even innovative. Perhaps some will lend themselves to your repertoire. In any event, see how concepts for producing efficient code can also relate to producing code efficiently.

Coding: It is unfortunate that many students are still taught to write programs with a pencil, first. It is understandable that logistics is sometimes a factor; homework assignments for those that do not own a computer have to be reckoned with. It is not so understandable that how to do live coding properly is seldom stressed. Old fashioned notions that coding on the fly is synonymous with "spaghetti code" is still preached in some schools. Live coding is the way we all do it today. When we adhere to time-tested disciplines, well organized programs are a natural byproduct of our methods.

Line numbering: All programs are numbered beginning with 1000, in increments of 10, from there on. While editing, additional lines can be inserted using the low-order digits from 1 to 9. When the need for more space arises, RENUM 1000 is done.

RENUM 1000 is a constant and frequent habit; it is always done just before a SAVE. RENUM is itself a handy quality assurance

tool; obviously it is an easy way to detect "unreferenced line" errors. Equally obvious, it is more efficient to resolve such errors immediately, during coding, rather than at run time.

For those reluctant to renumber a program, because you prefer to memorize GOTO and GOSUB addresses, consider the coding disciplines suggested below. They require no brain strain at all. And, they support being able to RENUM often. As we shall soon see, these methods simplify code swapping among programs as well.

The first line of a program (1000) is always, and only, a remark line. It contains a full-text version of the name of the program (i.e., the phrase from which the acronym-form of the file-name was derived). The release date and version number for this program is also maintained up-to-date on this top line. This is also where copyright notice is given, when applicable, to conform to the rules about submitting listings with copyright applications.

The second line of a program (1010) is always (and usually only) a GOTO, to the logical beginning of the program itself.

The third line of a program (1020) is always (absolutely) a SAVE statement with the file name of the program hard-coded as a quoted literal. This SAVE statement is immediately followed by a LIST-1020.

Here is a real-life example of what has just been described:

```
1000 'Payroll Ops W-2  1.0  08/22/89  (c) ACT-1,Inc. 1989
1010 GOTO 1200
1020      SAVE "POPW2":LIST-1020
```

The subtle advantage of line 1020 deserves explanation. While editing, SAVE is done very frequently to guard against time losses due to failures. (Mine, and the power company's.) It is easy to make typing mistakes. Every time we type SAVE, with a name, we increase the odds of unwittingly creating a second file with a slightly different name. Far worse, it is all too easy to accidentally overwrite another program with the one now being worked on.

So, once a program's name has been hard-coded into the program itself, from then on, merely typing RUN 1020 prevents spelling mistakes. The follow-on LIST stops execution, and it provides visible feedback, and a reminder that the release date or

version number may need updating. (Even the truly lazy that like the editor's function-key scheme can use this trick; just substitute RUN 1020 for one of those several useless strings we get for free.)

Line 1020 has a second purpose: Semi-automatic back-ups to a second disk drive. Immediately after a RUN 1020 the cursor will already be in a handy position. Cursor up one line, hit the space bar four times, and tab twice. If the program's name is properly positioned, we need to merely hit the insert key, then type in a drive letter and a colon, and hit enter.

Obviously, your own inclination may be to arrange the contents of the first few lines of a program differently. Notice the advantages, however, of version-control information up front, and, a built-in SAVE-name line always coded on a line number forever dedicated to just that purpose.

Before rambling on, see my inclination for depending on habits. This permits concentration on program logic; the mundane can depend on our automatic reflexes. This theme permeates all of what follows as well. Having said it once, here, it is no longer necessary to repeat what is so patently apparent.

The first three lines of a typical program from my library were shown above; what follows is a continuation of that same one. Notice that lines 1030 through 1190 are merely a branch table. And they are a table of contents. This table is also a modules map giving the current entry point into each module. The end of a block can also be easily discerned; it is the number of the line immediately preceding the first line of the next block of code, i.e., that line number minus 10.

```
  1030 GOTO 1310 'trapE       ERR error handler
  1040 GOTO 1400 'openQ  1    keyboard & init Q-vars
  1050 GOTO 1460   'Qpick       get menu selection
  1060 GOTO 1790   'Qmask       display menu or help pages
  1070 GOTO 1970 'openM  2    monitor & init M-vars
  1080 GOTO 2120   'Mread       get empl ID into M(1-8)
  1090 GOTO 2260 'openY  4    payroll files & init Y-vars
  1100 GOTO 2440   'Yread       get employees into ssan-tank
  1110 GOTO 2580 'openV  5    Vdisk prep & init V-vars
  1120 GOTO 2670 'openW  6    W2 flex disk output
  1130 GOTO 2780   'Wline       write output line/record
  1140 GOTO 2820     'Acode       transmitter = 1A & 2A
  1150 GOTO 3010     'Bcode       authorization = 1B & 2B
```

```
1160 GOTO 3210     'Ecode       employer ID = 1E & 2E
1170 GOTO 3420     'Wcode       W2 wages = 1W & 2W
1180 GOTO 3800     'Scode       supplemental = 1S & 2S
1190 GOTO 3830     'Icode       intermediate totals = 1I
1200 'begin
```

Once again, there is more to be seen here than is conspicuous. Much more. The simple use of LIST-1200 is an obvious and easy substitute for memorizing where things are. When a RENUM is done, this table of contents is automatically updated. Even as new modules are added. Line 1010 always tells us how far to LIST, to see this map, and where the program actually begins.

There are several more advantages to be derived from the above scheme, but, we should first dispense with a couple of side issues. One concerns run-time efficiency; the second considers presentation incompatibilities between interpreted and compiled programs.

Performance: One module can only call another via GOSUB. It always refers to the GOTO-line for that block, in the branch table (like the one shown above). No branching is ever done, directly, to another block's physical address; only logical addressing is permitted. GOSUB, ON ERROR and the like, must do indirect addressing via the branch table.

As described more fully elsewhere, GOSUB triggers a stack entry so the interpreter can remember where to RETURN to. On the first instance of any line-referencing statement, a search is made for the target line. Then, once found, that line's actual address in memory replaces the hard-coded line number in the referencing statement. And that is pretty efficient.

The use of a branch table is not purely efficient, obviously. It causes the interpreter to have to do two searches instead of just one. Plus, it effectively takes two statements to do what could be done with only one.

Because line searching has to be done only once per RUN, the overall performance impact on an otherwise efficiently coded program is pretty small. The time-cost for a back-to-back GOSUB/GOTO--once physical addresses are in place--is so small it would be hard to measure on even older and slower micros. The sum of all of this is small enough to be ignored in favor of the greater benefits to be derived from a well structured program. This is true for those of us that do not appreciate

being called spaghetti-coders, anyway.

QuickBASIC: As we well know, compilers are ignorant of the logical intent of programming statements beyond what they can discern from static semantics. They must depend on how lines are physically presented, whereas an interpreter can follow logic-routing paths. By quickly reviewing what has been suggested above, we can see that nothing was said that is contrary to either.

The suggestion for hard-coding a SAVE in line 1020 is one minor nuisance when moving from GW-BASIC to QuickBASIC. The compiler does not understand SAVE, or LIST, but neither will it ignore them. The obviously simple solution is to convert that line to a remark just before attempting compilation; merely insert an apostrophe between the line number and the word SAVE.

Because the initialization section of a program is always the first block of statements immediately following the branch table, the compiler has no problem with "processing" all of those GOTO statements before it gets to the real meat of a program. The compiler simply builds tables for procedural addresses, internally, similar to those for data addresses. Because the rest of a program is physically located beyond the initialization block, which comes physically just after the branch table, everybody is happy. We can easily read our programs, and, so can either the interpreter or the compiler.

With those side issues now aside, we can revert to the major advantages of using a branch table in the first place: Cut and paste tricks, and simplified debugging. Both of these relate to other methodologies as well, so, the business of swapping code is dealt with first, completely. How the branch table can also serve as merely one of several debugging aids comes later.

On occasion, MERGE is one obviously useful facility for splicing and grafting code segments. But, it is slow and cumbersome to use. Consider as a first choice an easier alternative of using SAVE, LOAD, RENUM, and line number editing tricks. Like this:

Use RENUM to open-a-gap where a paste-up from another program should be physically implanted.

SAVE the current program. (Remember, RUN 1020.)

LOAD the program that contains the piece of code wanted.

LIST the inclusive lines desired.

LOAD the program being edited (the one that was just saved).

Carefully edit the line numbers of the piece of code still shown on the screen. Be sure to hit Enter after each line number is changed; be sure to not hit Enter until a line's number is in fact changed to correspond to its intended location in the program currently in memory.

Now see the reason for a uniform scheme of line numbering in all programs: All line numbers are always four digits. It makes editing easier, and, it benefits changing internal references once pasted-up lines have been stored by the use of Enter. (Because of the discipline of always doing RENUM before any program is saved, it is easy to see what adjustment is needed for internal line references in new paste-ups.)

PS: Four digit line numbers is sufficient, even when always incremented by ten. No single program in my library ever gets close to exceeding a thousand lines.

Chapter 10 mentioned two aspects of coding style that should be looked at again here. One dealt with having, effectively, a "table" of variables because they are all named in the initialization block of a program; that table should be kept current, on the fly, as coding proceeds. In a similar fashion, the branch table should also be kept up to date. (See how one habit tends to reinforce the other, also.)

When a new block of code is built, by whatever means, its table entry should be inserted at that same time. Now, RENUM takes care of everything else, automatically.

That other suggestion about programming style was: Include the nickname for a block, as a remark, at the end of any line that addresses another block. Notice that nicknames are also a feature of the branch table (and also reflected in the first line of a block), and, see how all of this relates to keeping track of where things are. And how RENUM is a mechanical tool: Line numbers are a physical issue, only incidentally related to the logical make-up of a program.

It should also be obvious by now what the real advantage of

a branch table is, beyond that of its aesthetic worth: Blocks can be moved in a program, and between programs, easily. If, on the other hand, calls were permitted to physically address other blocks of code directly, more time would have to be spent finding and changing all such references. The branch table localizes all calls external to any block. Although numerous calls may be made to a given block, if it is moved or replaced, changing its call-line destination address in the branch table will keep all callers happy, automatically, no matter where they are calling from.

There is one more small trick closely associated with all of this business about line numbers. Unknown addresses should be coded as 9999. Often while we are doing live coding, we can think faster than we can type. (We know a jump is needed, but we haven't yet written that part, for example.) Another case that is similar is when we are creating blocks to be merged into a program in a separate step, later. RENUM will flag all 9999-stubs for us, and tell us where they now are.

A second aspect of reusing old code is renaming variables so that they conform to current needs. It is somewhat amazing that GW-BASIC still has no built-in facility for this. (All other modern language products can do it; even, QuickBASIC.)

The QuickBASIC editor can in fact be used for this purpose, but so can nearly any word processor. In fact, my preference is for using my favorite word processing program (PC Write) because it is a much better editor than QuickBASIC's. In either case, obviously, SAVE-as-ASCII must be done, and that is always a tiresome chore.

> PS: One of the simple own-code tools in Chapter 15 is handy for at least finding variables quickly so that we can edit them manually without jumping in and out of BASIC.

Only a few more notes are needed now about reusable code. My own argot has crept in elsewhere, but a few terms should be defined here, specifically.

> Freeze-dried code is a phrase for tightly condensed blocks that seldom needs logic modifications no matter where they are used. They also tend to avoid any internal GOTO-type of statements. And they use the minimum number of variables possible; especially those of a global nature. Because they do not anticipate modifications, they seldom contain any remarks, either.

Canned code is a little more liquid than freeze-dried; these can mostly be used in bulk, but may require minor alterations in logic when reused in different situations. They are best constructed such that, even if they resort to internal line referencing statements, it is easy to see those connections. One or two-space indenting patterns and a few remarks can help make it easy to reuse blocks such as this.

Prefabricated modules are stored in "library programs" that are contrived to be simply a repository of reusable tricks. Such as those included in Chapter 14 of this book. Notice all of those pieces have preamble remark lines that can be read for research, but discarded after a module is fully transplanted.

Retread code is the most flexible of all: Code swiped from another program, in bulk. Chapter 13 contains many examples of this type. They tend to be user-task or technique-driven, seldom transportable generic processing algorithms. Even so, it often takes a lot less time to edit large blocks of code swiped from an old program than it would to recode it all, all over again. Equally, far less effort is needed to reuse old ideas that worked well before; especially for those that are boring, or, undeserving of an intellectual investment of our valuable time.

Transient tools: Chapter 15 contains several of these. They are self-contained modules that can be merged into the fabric of a program during its development, but deleted from the end product. Transients may be generally useful, predesigned tools, or small custom creations for use in a single program, or for several, in an application set.

Reflecting on all of the ideas suggested thus far about the overall fabric of a program, see now also the worth of a branch table during development and debugging: It is easy to reroute any given branch so that it can temporarily flow through a transient tool-routine. All that is needed is to insert GOSUB in the table, in front of a GOTO.

Jumping to a GOSUB-tool before a GOTO (to a target destination) is a before-the-event trap, of course. For an after-the-fact trap, simply change the GOTO itself to a GOSUB, and follow it with a GOTO, to any transient-tool that ultimately ends itself via RETURN.

So, spaghetti-writers like TRON and TROFF. We that have more method in our madness do it differently, and more efficiently. It is unnecessary to "trace" a well organized program; where something is done should be self-evident by reason of how the major tasks are mapped, and accessed. Traps inserted in the branch table can be routed to transient-tools to take snapshots of what is in variables before and after, to isolate what is going on (right or wrong), without affecting what is being displayed.

The branch table is also a handy place to implant other types of quality assurance and diagnostic aids: Small watchdogs need not be full-fledged elaborate subroutines, obviously. STOP is a natural one, of course. So is FRE.

Recall all of those admonishments in Chapter 4 about coding so that unexpected time-outs should never occur because of clutter in your "free" string-space area. After a program has been fully initialized, insert a LOCATE and PRINT FRE(0) at a choice place in the branch table and watch that feedback. In a well behaved program the answer should remain constant; if it does not, some variable has been used that was not pre-named during initialization.

Another simple watchdog that can be inserted for a while in the branch table is a couple of statements that will display usage counts. LOC is a good one for monitoring file processing; it is especially useful during early testing phases to preclude runaways that might inadvertently fill up a disk with junk.

The location of the branch table itself can be a significant aid. Down in the bowels of a program we know that no GOSUB should have a "big" line number; they are supposed to jump to the branch table. Because those lines are at the top of the program, a GOSUB aimed at a high-order line number may be the cause of an "Out of memory" error. (Due to a stack overflow, as described in Chapter 10.)

Because of having localized all jump addresses into a single branch table, it is also easy to identify who is calling whom, from where: Simply delete a branch table entry and do a RENUM. All callers will announce themselves, and where they are coming from.

To get accurate feedback, and to keep from having to retype the table entry in question: Duplicate the table-entry line number by overtyping the low-order 0 with a 1 (then hit Enter). Now delete the real GOTO-line. Not only will RENUM correctly report all lines that contain jumps to the missing branch table line, the table will be rehealed automatically. (If line 1090 is renumbered as 1091, for example, RENUM 1000 will change 1091 back into 1090 for you, for free.)

Having now seen several ways that RENUM can be exploited as a programming aid, here is a final tidbit about line numbers: As the interpreter strums along, it saves the number of each line that is being executed. By knowing where it is holding this little gem, we can look at it whenever the need arises.

See this: L = PEEK(47)*256+PEEK(46). The pair of bytes at decimal address 46 and 47 (and a default DEF SEG) is where the interpreter remembers what line number is currently executing. When you STOP, or do a break, the number that is displayed at that time comes from this location, up in the interpreter's own working storage area.

When debugging logic-flow problems a simple expression like the one just shown can be inserted at strategic locations. After a brief trial run, the trap-variable (L) can be examined as an aid in determining whether specific lines were ever executed.

Another way to find line-addressing bugs is to use one of the tools listed in Chapter 15. A complete GOTO-from-where listing of all line references in a program can be gotten by using the tool called LXREF (Line Cross-Reference). A variation of that tool is called LHITS (Lines Hit Selectively).

The LHITS tool lists only those lines that have been referenced by an in-line statement during execution. By stopping a trial run at a strategic point, and running LHITS, it can easily be determined which branch-paths have actually been taken up to that time.

The other two tools listed in Chapter 15 are useful for solving problems associated with variables. One simply lists all of the variables named in a program, and the lines in which they occur. It is called VXREF (Variables Cross Reference).

The tool called VLIST (Variables List) is especially handy for

optimizing efforts. Recall how the order of presentation of variables in a program can sometimes have a significant impact on run time performance. (Viz Chapter 3.)

After a trial run has been fully initialized, the VLIST tool will list all of the variable names that the interpreter has set up in working storage, in table-search order. By relating that arrangement to what variables are used for, a program can be fine-tuned to ensure the most frequently used variables are forced to be near the top of the search list.

VLIST can also be used as a traffic cop. Suppose AZ is not the name of a real variable in a program. (VFIND can be used to make sure of this.) In an area where logic-flow seems to be suspect, implant a temporary AZ = 1. Run the program. Stop. Run VLIST. Whether AZ is listed, or not, will reveal what was (or was not) encountered while branching to and fro.

The homemade tools described thus far are used mostly during debugging exercises. There is another type of tool useful for similar purposes: QuickBASIC. The interpreter can only help find programming mistakes in statements and expressions that it bumps into "dynamically". A compiler, by definition, is a static code analyzer. It scans everything and can thus detect some coding errors that the interpreter may never get to see.

When we pour a GW-BASIC program through the compiler it may erroneously flag many things that it thinks are errors. When we know the interpreter knows better, we can ignore what the compiler says. It is sometimes surprising, however, what can be found by attempting to compile programs written expressly for use with the interpreter. (It can be humiliating, even, to have QuickBASIC spot real errors in programs that you knew, absolutely, were bug free.)

There are times when QuickBASIC programmers can make good use of GW-BASIC. There is one situation even, that practically demands the use of the interpreter as a platform for developing compiled programs: QuickBASIC is a memory hog; it cannot be used when a large part of memory has been allocated as VDISK (aka RAMDRIVE).

What has been covered thus far are suggestions about how to structure a program, and how to easily fabricate programs from pieces of existing code. And some ideas for debugging, with

emphasis on using homemade tools. The following tricks from my bag are more like jigs, than tools. They are useful when building screens and menus.

The editor in GW-BASIC allows us to wander around on the screen and type anything, anywhere. It makes no attempt to "read" what we are seeing until we hit Enter. When we do hit Enter, all that is looked at is the line the cursor is on at that point. So, it is possible to draw a picture, and BSAVE it, without ever having to exit BASIC.

Drawing boxes is tiresome, however, when done with codes and the Alt-key. And it is not easy to remember all of those codes for the different box-characters. The following two primitives are useful for drawing with the cursor arrow-keys. They can be used as GOSUB-tools, as is. They are also easy to modify for different screen modes. (They can also be used as a basis for building a full fledged, customized, mass production screen editor, PS.)

The first subroutine is for drawing line-boxes. It makes use of the numeric keypad as a mnemonic template. Seeing those keys as a square, 7 is an upper-left corner; 3 is a bottom-right corner. And 5 draws a center-intersection. And 4 for example, draws a vertical line with an intersection coming from inside the box. And so on.

The letter-keys S, D, V, and H set or change the "style" of box that can be drawn. S and D mean single, or double lines. V means double-lines vertically, but single horizontally; H is the opposite, double horizontal, with single vertical lines.

The insert-key is a toggle switch. When on, the number keys (with Num-Lock on) print corners and intersections, and the cursor keys draw lines in the direction indicated. When the insert-key is toggled off, the cursor may be repositioned on the screen without drawing anything. The space bar can be used for erasing mistakes. Exit is via the Esc-key.

```
7000 'Ldraw
7010 DEFINT C-L:DEFSTR Q
7020  Q=MKI$(0):L=CSRLIN:C=POS(0):G=0:I=0:D=1
7030 WHILE G-27:LOCATE L,C,1
7040  LSET Q=MKI$(0):WHILE Q=MKI$(0):MID$(Q,1)=INKEY$:WEND
7050  G=ASC(Q):F=ASC(MID$(Q,2))
7060  H=INSTR("SDVH",CHR$(G))+INSTR("sdvh",CHR$(G))
7070   IF H THEN D=H:F=0:G=0
```

```
7080   IF F=82 THEN I=I XOR 1:F=0:G=0
7090   K=INSTR("123456789HPKM",CHR$(F+G))+1:H=SGN(K)*D
7100  IF H=1 THEN K=ASC(MID$(" @AYCE4ZB?33DD",K))+128 'single
7110  IF H=2 THEN K=ASC(MID$(" HJ<LN9IK;::MM",K))+128 'double
7120  IF H=3 THEN K=ASC(MID$(" SP=GW6VR7::DD",K))+128 'dbl vert
7130  IF H=4 THEN K=ASC(MID$(" TO>FX5UQ833MM",K))+128 'dbl horz
7140  L=L+(L>1 AND F=72)-(L<25 AND F=80):IF G=32 THEN K=32
7150  C=C+(C>1 AND F=75)-(C<80 AND F=77):IF F THEN LOCATE L,C
7160  IF G+F*I THEN PRINT CHR$(K);
7170 WEND
7180 RETURN
```

The following shorty is a simple dupe-key routine. When Enter is hit, it "captures" whatever character the cursor is resting on. The insert-key toggles drawing mode on or off. When on, the cursor arrow keys "dupe" the character that was captured by Enter, and move in the direction indicated. When drawing mode is turned off, the cursor arrows work as they normally would. Again, the Esc-key is the exit key.

```
8000 'Qdraw
8010 DEFINT C-L:DEFSTR Q
8020  Q=MKI$(0):L=CSRLIN:C=POS(0):I=0
8030 WHILE ASC(Q)-27:LOCATE L,C,1
8040  LSET Q=MKI$(0):WHILE Q=MKI$(0):MID$(Q,1)=INKEY$:WEND
8050  G=ASC(Q):H=G:F=ASC(MID$(Q,2)):IF F=82 THEN I=I XOR 1:H=0
8060   IF G=13 THEN K=SCREEN(L,C)
8070   IF G>31 THEN F=77 ELSE H=K
8080    L=L+(L>1 AND F=72)-(L<25 AND F=80)
8090    C=C+(C>1 AND F=75)-(C<80 AND F=77)
8100   IF I*H THEN PRINT CHR$(H);
8110 WEND
8120 RETURN
```

The above shorties can easily be used as transient tools. When editing a program, do a direct-execution GOSUB on a line that is unused at that point. Draw whatever, then move the cursor to a position where the "Ok" will not irritate you when you hit escape. (As we all know, RETURN from a directly executed GOSUB returns control to the BASIC editor.)

It is sometimes handy to keep the above two subroutines in your tool kit, even when you have spent time developing a grandiose screen builder. Consider, for example, a menu program that has self-maintenance capabilities. Or a report-writer that lets an operator draw pretty boxes.

One final shot: Reusable code should be fundamental to our methods for building programs in BASIC. Programs whose primary structure is based on a branch table are easily debugged, and, they can be economically maintained.

Old timers will undoubtedly notice that nothing suggested here is really new, at all. Branch table concepts date back to the earliest days of programming. And way back when, we all kept old decks of punched cards that contained our favorite pieces of canned code. All that has been remarked upon here is some ideas on how to bring those old, timeworn concepts up to date.

The iron is dramatically different today, but what we have to do as programmers is fundamentally the same as it has always been. A mouse may be a clever device to some, but the essence of efficiency is a function of our methods, not our gadgets.

Perhaps some will find some of the above useful; it has been the basis of my methods for a long time, long before things like a mouse or a monitor were ever invented.

Chapter 13 = TECHNIQUES

Cleverness is undoubtedly a core personality trait of people who like to program. We enjoy devising clever solutions to problems. Ego plays an important part as well; we tend to think we are more clever than most. Most of the time, anyway.

Having no knowledge of psychology, there is also a confusing aspect to our makeup: Seldom can anyone else see how clever we are. Users may favorably comment on our products, yet all they can see is what we have made visible. Little do they know what is really happening inside a program.

Compliments from lay operators are meager fuel for our egos. When they offer criticism, a defensive impulse is triggered. How can anyone take potshots at what they cannot see, or, when they cannot even begin to understand the intellectual feats of our fait accompli.

The above thoughts seemed appropriate preamble, to me, for a chapter that offers ideas about programming technique. It is a natural tendency, for most of us, to view rather skeptically what another programmer thinks is clever. After all, we are clever too.

While skimming what follows, look for the cream, and excuse any unintended braggadocio with clenched teeth. You can freely use any gems deemed worthwhile, with impunity. No one will ever know that you actually stole an idea from a book. Equally, my ego is in no jeopardy; whatever you might guffaw at will remain unknown to me. When you opt to use superior techniques of your own, savor your cleverness. Either way, what is offered here is meant to be food for thought. Healthy programs are well fed programs; it matters little where the beans come from, assuming they were grown in fertile soil in the first place.

Touch-typists like to keep their fingers on the home keys. The following menu makes use of initial-letter selection, for just that reason. And it matters not whether caps-lock is on or off. Which is good technique.

As programmers, we appreciate the worth of mnemonics. So will our users. It is obviously easier to remember that S means Search, for example, than F1 or F7, or something. And few of us have fingers long enough to reach the function keys without

exaggerated hand movement.

Because data entry operators have to constantly shift their eyes back and forth between input documents and the monitor, we should make it as easy as we can for them to memorize the keys used for menu selections. Experienced operators do not "read" menus (nor look at the keyboard). In actual practice, they seldom read anything on the screen; they merely glance up now and then to confirm they are in step with what the program is doing.

Video highlighting is one way to help an operator spot things on the monitor quickly. Easy pattern recognition is another. Color-bar highlighting of menu lines can be seen as steps on a ladder. After a little practice, few users actually read what is printed on a rung. They merely confirm the step they are on.

The menu driver that follows is attuned to all of that advice from the human factors engineers, and it goes one step farther. It also lets those that prefer, to move the color-bars with the cursor arrow keys and make selection of a highlighted choice with the Enter key. And it does it all fairly efficiently in interpreted BASIC.

Code needed during program initialization:

```
  2000 DEFSTR M-Z:DEFINT C-L
  2010 Q2=MKI$(0)                'Q2Keys
  2020 Q1=CHR$(0)                'Q1Key
  2030 DM=5                      'DoMenu
  2040 BM=&HB000                 'BaseMonitor (mono)
```

A "main menu" display-and-select subroutine:

```
  2700 'menuM
  2710 CLS:LOCATE 4,1,0:LSET Q1=CHR$(13)
  2720 PRINT ,, " Keys index  ";Q1
  2730 PRINT ,, " Name scan   ";Q1
  2740 PRINT ,, " SS# search  ";Q1
  2750 PRINT ,, " Add record  ";Q1
  2760 PRINT ,, " Posting     ";Q1
  2770 PRINT ,, " Tax changes ";Q1
  2780 PRINT ,, " Run reports ";Q1
  2790 PRINT ,, " Disk jobs   ";Q1
  2800 PRINT ,, " Quit POPS   ";Q1
  2810 L=DM*2+2:DM=0                          'last DM line
```

```
 2820 DEF SEG=BM                                        'video RAM
 2830 WHILE DM=0:C=(L-1)*160+57                         'col(29*2-1)
 2840  FOR I=C TO C+24 STEP 2:POKE I,112:NEXT           'color 0,7
 2850  LSET Q2=MKI$(0)                                  'clear kb
 2860  WHILE CVI(Q2)=0:MID$(Q2,1)=INKEY$:WEND           'get a key
 2870  FOR I=C TO C+24 STEP 2:POKE I,7:NEXT             'color 7,0
 2880  LSET Q1=Q2:I=INSTR("HP",RIGHT$(Q2,1))                 'up/down
 2890  L=L+2*(I=1 AND L>4)-2*(I=2 AND L<20)                  'cursor
 2900  DM=INSTR("KNSAPTRDQ",Q1)+INSTR("knsaptrdq",Q1)        'letter
 2910  IF ASC(Q1)=13 THEN DM=(L-3)/2                        'enter
 2920  IF I+DM=0 THEN SOUND 99,3                            'oops
 2930 WEND:DEF SEG
 2940 RETURN
```

Several variations can be derived from the above. Paint-time with PRINT is never very fast, no matter how it is done. One better alternative is to BLOAD the screen. Sometimes. The keyboard driver above, from line 2810 down, can be used either way.

See also how a basic technique is incorporated for the menu to "remember" what selection was made last. By respecting DM as a global-variable, when a return to the menu is done, the line that is highlighted indicates where we are returning from. DM is a number--1 to 9 in this case--that is used initially in an ON GOSUB as a major-task dispatcher. It can also be used thereafter for mode testing, in subroutines that may be called from more than one major task.

There is also an opportunity here to influence an operator's choice about what to do next, by preloading DM with a default selection. If they agree with what they see, they merely have to hit Enter.

Another good technique, at times, for you and the operator is to anticipate. If the Esc-key is the menu-trigger for example, a pretest can be done using a compound INSTR expression like the one in line 2900, to save having to jump to the menu at all. Which saves you both time. (A la, "hot keys".)

Performance-wise one of the fastest functions in GW-BASIC is INSTR. It is especially fast when the string to be examined is a hard-coded literal, as typified in the keyboard driver above. (Remember the overhead involved when the interpreter has to search for a variable, viz Chapter 3.)

Even when three or four variables are involved, INSTR can still often be a solid quoin for functional structures. See how it is a keystone in the following routines that are useful for a particular type of ISAM. (Assumes DEFSTR M-Z and DEFINT C-L.)

```
3000 'openFV
3010 FV=4:OPEN VI AS FV LEN=250             'VI = index file
3020  FIELD FV,250 AS VB:GET FV             'VBuffer; 1st GET
3030  FIELD FV,10 AS V1,230 AS V2,10 AS V3  'Vfields
3040 IF LEN(V0) THEN 3110 'already initialized
3050  V0=V1:V4=V3          'shift-out buffers
3060  VF=LEFT$(V1,8)       'VFind = search-for key
3070  VG=VF                'VGot = last key found
3080  CV=INSTR(VB,VF)      'ColV = buffer position
3090  LV=LOC(FV)           'LocV = buffer record #
3100  GV=0                 'GotV = target record #
3110 RETURN

3120 'Vget
3130 CV=INSTR(VB,VF):IF CV THEN 3180                  'in buffer
3140  WHILE VF<VB AND LOC(FV)>1:GET FV,LOC(FV)-1:WEND 'back up
3150  WHILE VF>V3 AND V3>" ":GET FV:WEND               'forwards
3160  LV=LOC(FV):CV=INSTR(VB,VF):IF CV THEN 3180      'exists
3170  CV=1:WHILE VF>MID$(VB,CV) AND MID$(VB,CV)>" ":CV=CV+10:WEND
3180 GV=ASC(MID$(VB,CV+8))*100+ASC(MID$(VB,CV+9))-10100
3190 LSET VG=MID$(VB,CV)
3200 RETURN

3210 'Vchg
3220 GOSUB 9999:GOSUB 9999 'Vget:Vdel; preload VF with old key
3230  LSET VF=Q8           'Q8 is holding replacement key
3240 GOSUB 9999:GOSUB 9999 'Vget:Vadd
3250 RETURN

3260 'Vdel
3270 L=LEN(VB)-10:MID$(VB,CV)=MID$(VB,CV+10,L)     'overlay
3280 MID$(VB,11)=LEFT$(VB,L)                       'shift right
3290 PUT FV,LV:LSET V4=STRING$(10,0)               'unload
3300 FOR I=LOF(FV)/LEN(VB) TO LV STEP-1:GET FV,I   'from bottom
3310  LSET V0=V1:LSET VB=MID$(VB,11,L):LSET V3=V4  'shift left
3320  PUT FV,I:LSET V4=V0:NEXT:GET FV,LV           'rewrite
3330 RETURN

3340 'Vadd
3350 LSET V0=VF                                    'key entry
3360 MID$(V0,9)=CHR$(GV\100+100)                   'rcd pointer
```

```
3370 MID$(V0,10)=CHR$(GV MOD 100+100)              '2nd byte
3380 LSET V4=V3:L=LEN(VB)                          'shift out
3390  IF CV<L-9 THEN MID$(VB,CV+10)=MID$(VB,CV,L)  'make slot
3400 MID$(VB,CV)=V0:LSET V0=V4                     'insert
3410 WHILE V3>" ":PUT FV,LOC(FV)                   'until end
3420  GET FV:LSET V4=V3:MID$(VB,11)=LEFT$(VB,L-10) 'shift right
3430  LSET V1=V0:LSET V0=V4:WEND                   'shift in
3440 PUT FV,LOC(FV):LSET VG=VF:GET FV,LV           'rewrite
3450 RETURN
```

The above ISAM routines are from a custom accounts payable application. They do full index management in the program that provides for operator maintenance of the vendor's file. That application has the following functional requirements:

Total active records on file is not expected to exceed more than a few thousand.

Additions and deletions are somewhat infrequent; generally, not more than a few of either are ever done at one time.

Duplicate keys are not permitted, but, the spelling of keys that do exist may be changed at any time. (Keys are seldom changed in actual practice, however.)

Record keys may be from 1 to 8 characters, and may be any combination of upper case letters, digits, and symbols in the code range 32-90, decimal, but a key may not begin with a space-character.

The mechanical principles of this ISAM scheme are:

Keys are maintained in alphabetically ascending sequence.

New keys are in-sorted when new records are created. Keys for deleted records are removed from the index, and the index is shortened accordingly, but, the target record remains in situ in the master file (for historical reporting purposes).

The overall length of the index file grows with new additions, but only when such additions exceed the original length of that file. The overall length of the index file is never shortened; space once taken up by keys that are deleted wind up as hex-zero strings at the bottom of the index (so that an "old" index-entry slot can be reused for future additions).

The system performance issues related to this scheme are:

Master file records are large (512 bytes).

Several other files must be open at the same time.

To keep disk thrashing to a minimum, the fewer the number of accesses needed into the index file, the better.

By maintaining the overall physical length of the index file (and the master) as constant as possible, fewer noncontiguous clusters will be used. When compress-type operations are done, "old" clusters will be forever contiguous. Meanwhile, only the newest clusters allocated will be displaced from the rest.

By reason of the index file being separate from the master, and because each index-entry is a mere 10 bytes, up to 51 entries can be grabbed into a DOS buffer on a single GET. By keeping only active-record keys in the index--compressed toward the top of the file--key searching and sorting need not be hampered by keys associated with dead records.

Implementation of the above code assumes the following:

Each of the four modules are called on as needed.

The 9999-stubs in the Vchg (Vendor Change) module must be readdressed to point into the branch table, so as to call the other subroutines in the order indicated. (Chapter 12 covers the branch table business in detail.)

The VF variable (Vendor Find) must be preloaded with the key of a record to be found before Vget is called.

The content of VF must be quality-checked before it is passed to Vget.

VF can be compared to VG (Vendor Got) after a search to see if a match was found. If not, VG will contain the key of what would be sequentially next, if the search key did in fact exist. (And LV and CV are already positioned for a call to Vadd, to in-sort a new addition, or a changed spelling of an old key.)

VG will contain the first key in the index, or the last one, if the search key is lower or higher in sequence than are those already in the index.

Vget can be used on a trial basis, to guard against attempts to coin new keys that would duplicate already existing ones.

GV must be preloaded with the relative record number of a new master record, and VF must contain its key before Vadd (Vendor add) is called. Which also means GV must contain the relative record pointer-portion of keys being repositioned because their spelling is being changed, when successively calling Vdel and Vadd from Vchg. (A not very fast, but, cheap technique for a seldom used capability.)

Now see the role of INSTR as fundamental to the above ISAM subroutines: On a GET, 10 keys are pulled into a 250-byte string. A simple INSTR can search that buffer, quickly, to see if it contains a given 8-byte record key. If it does, the two bytes immediately following the matching key contain a relative pointer for that associated record in the master file.

If a search fails, if the key being looked for is less than the left-most one of the index record buffer, we can walk backwards in the index file. When the search key is larger than the right-most key in the buffer-string, a forward search can be done. Each test is effectively "paging" in increments of ten; physical disk accesses are thus at least one-tenth of what they might be, otherwise. Theoretically, at least. (GET in BASIC only equates to a physical read if a needed record is not yet in a DOS buffer, as we already know.)

Empirical experience on old 4.77 MHz machines results in an average find-time in the neighborhood of 75 milliseconds, with about 1500 active keys stored on a hard disk that claims an average access time of 35 MS. Not too shabby for a GW-BASIC program; it can even compete favorably with some of the new 4GL products that like to denigrate us poor BASIC folks.

The two-byte record pointer that follows each key is specially contrived to ensure that INSTR does not confuse "binary values" with characters contained in keys. Lines 3350 and 3360 in Vadd encode record pointers; line 3180 in Vget decodes them.

Because valid characters in keys are restricted to codes less than 100 (decimal), adding 100 to each of the two bytes that represent record pointers ensures that INSTR will always align

on keys properly. This does mean that the largest relative record number that can exist in the master file is 15,599. But it also means that the index file can be looked at as ASCII text. Like with the DOS TYPE command, for example, while doing testing and debugging.

A secondary advantage to the way key entries are stored in the index described above is that the index file may be redefined for the benefit of other programs. Such as:

```
OPEN VI AS 1 LEN = 10       'VI = index file
FIELD FV,8 AS VK,2 AS VL    'Vendor Key, Vendor Location
```

During posting, for example, a conventional binary search can be used in an alternative Vget (Vendor Get) module. Similarly, during report runs, the index can be read serially, as 10-byte records. A key beginning with CHR$(0) denotes the end of the index if LOF has not been encountered. See again how doing things the hard way, in BASIC, can also make it easy for some to be done the best way, depending on our needs at a particular point in time.

An alternative ISAM scheme can be based on MRI (Memory Resident Index) principles. For short key requirements--of 6 or fewer characters--a double-precision array can be used efficiently as the in-memory index tank. That entire tank can be loaded quickly using BLOAD. If any changes are made (to the index) it can be unloaded, quickly, at the end of the job with a BSAVE.

The following program exhibits these several techniques. It is also a "prefab program". Having this program stored as is, on a disk somewhere, a totally new application can be fabricated by making changes and additions to this skeleton.

```
1000 '6-byte MRI-ISAM skeleton
1010 GOTO 1070
1020      SAVE "MRI":LIST-1020
1030 GOTO 1360 'keyM          rebuild key index
1040 GOTO 1470 'newM          add new record
1050 GOTO 1570 'fixM          fix index (del/chg)
1060 GOTO 1660 'getM          locate record
1070 'begin
1080 DEFSTR M-Z:DEFINT C-L:DEFDBL A
1090  I=0:E=0:F=0:H=0                      'locals
1100 CLS:KEY OFF:FOR I=1 TO 10:KEY I,"":NEXT
```

```
1110  AM=2000                                  'Absolute MRI
1120  IM=1                                     'IndexM - A(IM)
1130  FM=1                                     'MasterFile#
1140  LM=80                                    'LenMasterRcd
1150  DEF FNEM=LOF(FM)/LM                      'EndMaster
1160  DEF FNGM=CVI(MID$(M8,7))                 'GetMaster
1170  M8=MKD$(0):M6=SPACE$(6)                  'MRIfields
1180  MI="testfile.mri"                        'MasterIndex
1190  MF="testfile.dat"                        'MasterFile
1200  DIM A(AM)                                'MRItank
1210 OPEN MF AS FM LEN=LM
1220  FIELD FM,6 AS MK,2 AS ML,LM-10 AS MD 'MKey,MLoc,MData
1230  FIELD FM,LM AS MB                        'MasterBuffer
1240  IF LOF(FM) THEN 1270                     'master exists
1250   LSET MK=CHR$(254):GOSUB 1040 'newM
1260   A(0)=1:BSAVE MI,VARPTR(A(0)),AM*8+8
1270 BLOAD MI,VARPTR(A(0))                     'load MRItank
1280  IF A(0)=0 THEN GOSUB 1030 'keyM (or enforce a restore)
1290 LSET M8=MKD$(A(1)):GET FM,FNGM            'first record

1300 '*************************************************
1310 '**            mainline code goes here            **
1320 '*************************************************

1330 RESET
1340  BSAVE MI,VARPTR(A(0)),AM*8+8             'rewrite MRI
1350 END

1360 'Mkeys
1370 IF SGN(A(0)) THEN 1460
1380 A(0)=FNEM                                      'size index
1390 FOR I=1 TO A(0):GET FM,I:A(I)=CVD(MB):NEXT
1400 L=A(0):H=(L-1)/2                               'sort index
1410 WHILE H:FOR I=1 TO H+1:E=1:WHILE E:E=0
1420  FOR J=I TO L-H STEP H
1430   IF MKD$(A(J))>MKD$(A(J+H)) THEN SWAP A(J),A(J+H):E=1
1440  NEXT
1450 WEND:NEXT:H=H\2:WEND
1460 RETURN

1470 'newM
1480 E=A(0):LSET M8=MKD$(A(E)):LSET ML=MKI$(FNGM)
1490  IF ASC(M8)=254 THEN 1510             'MK=new key
1500  A(0)=A(0)+1:E=A(0):LSET ML=MKI$(E)  'stretch file
1510 A(E)=CVD(MB):LSET M8=MB:PUT FM,FNGM  'on bottom
1520 FOR I=E TO 2 STEP-1                   'bubble up
1530  LSET M8=MKD$(A(I)):LSET M6=M8:LSET M8=MKD$(A(I-1))
```

```
 1540  IF M6<LEFT$(M8,6) THEN SWAP A(I),A(I-1) ELSE IM=I:I=0
 1550 NEXT
 1560 RETURN

 1570 'fixM
 1580 PUT FM,CVI(ML)                          'update master
 1590 FOR I=IM TO A(0)-1:A(I)=A(I+1):NEXT  'shuffle keys up
 1600  MID$(M8,1)=MK:A(A(0))=CVD(M8)       'put on bottom
 1610 FOR I=A(0) TO 2 STEP-1                'bubble up
 1620  LSET M8=MKD$(A(I)):LSET M6=M8:LSET M8=MKD$(A(I-1))
 1630  IF M6<LEFT$(M8,6) THEN SWAP A(I),A(I-1) ELSE IM=I:I=0
 1640 NEXT
 1650 RETURN

 1660 'getM
 1670 H=(A(0)):I=H\2:H=H+1:F=0              'search for MK
 1680 FOR E=0 TO 1:LSET M6=MKD$(A(I))
 1690  IF MK<M6 THEN H=I:I=I-(H-F)\2
 1700  IF MK>M6 THEN F=I:I=I+(H-F)\2
 1710 E=ABS(MK=M6 OR I=H OR I=F):NEXT
 1720  IM=I-((MK<=M6 OR I=A(0))=0)          'ISAM pointer
 1730  LSET M8=MKD$(A(IM)):GET FM,FNGM      'get master
 1740 RETURN
```

The above program is predicated on several design assumptions:

Record keys may be from 1 to 6 characters. Any character is permitted that has an ASC value lower than 254.

Duplicate record keys are not permitted.

Keys are dynamically maintained in alpha-ascending sequence.

The contents of a key may be changed.

Keys may be deleted; active keys are kept compressed toward the top of the index.

The maximum length of the associated master file cannot be greater than the DIM-argument for its index (but it can be less).

When a master file record is deleted, its key is changed by overlaying a CHR$(254) in the first byte, and the index is updated by down-sorting that key to the bottom of the index.

When new master records are added to that file, "old" deleted records are reused, if any exist, so as to minimize unneeded file growth.

Calls to the three primary modules must adhere to fairly simple rules.

1 - newM: Quality check the operator's "new key", then move it to MK and call getM. If M6 and MK are not equal, MK is not a duplicate, so, call newM to update the index, and to prewrite the associated master file record.

2 - fixM: This routine does double duty. It shuffles keys for both changed-keys, and deleted records. For either, move the existing key into MK and call getM. For a change, put the new key in MK and call fixM. To delete a record, the new key is the old one, modified by MID$(MK,1) = CHR$(254) before the call to fixM is made.

3 - getM: If the key in MK can be found, MB will contain the requested record, and IM will point to that key in the index. On a "not found", the record fetched (and IM) will be that of what would be next in sequence, if the requested key really did exist.

Some special advantages can be had with this basic scheme. On any GET, LOC(FM) should equal CVI(ML)--if it does not, the master file is probably corrupt. The BSAVE-index file can be used as an operations-integrity safeguard. On any first need to "update" anything, move A(0) to a hold-variable, set A(0) to zero, and BSAVE the index (then restore A(0) as it was). See line 1280. On every start-up a test is made to see if A(0) is zero, which would indicate an abnormal termination. (Line 1340 is for a "normal" ending. After RESET purges DOS buffers, the updated index is rewritten to disk, with A(0) indicating how many records are currently in the master file.)

That other functional routine included here (keyM) is for (re-) building an index. This is useful for "converting" an existing master file coming from some other source, or, for reconstruction of the index in the event that file has been clobbered or lost. To use keyM, set A(0) to zero, then jump to it via a GOSUB.

File integrity must constantly be a concern of any responsible programmer. Yet, in the wonderful world of DOS, the manuals

continuously treat this subject lightheartedly. Witness what the (IBM) DOS 3.3 Technical Reference says:

> "An application program should not concern itself with the way that DOS allocates disk space to a file. The size of a cluster is only important in that it determines the smallest amount of space allocated to a file at one time."

The (MS) DOS 3.3 User's Reference says:

> "You should run CHKDSK occasionally on each disk to check for errors."

It is not amusing that the manual for us technical-types says not to worry, but the manual for "users" says they should (but no definition is given for "occasionally").

The expensive manual does have a chart showing how to figure out cluster sizes for floppies. Nothing is provided anywhere, in the software or the manuals, to aid us in determining the size of a cluster on a hard disk. The manuals merely say one cluster is 1 or more sectors; the number of sectors is based on the size of a disk, and how it is partitioned. No more is said, however, about how it is actually computed.

Here is one clumsy technique for dynamically determining the size of a cluster:

```
1000 DEFSTR M-Z
1010 SHELL "chkdsk >junk.one"        'phase-1
1020 OPEN "junk.one" FOR INPUT AS 1
1030  FOR I=1 TO 7
1040   LINE INPUT #1,X
1050  NEXT:W=X
1060 CLOSE
1070 SHELL "chkdsk >junk.two"        'phase-2
1080 OPEN "junk.two" FOR INPUT AS 1
1090  FOR I=1 TO 7
1100   LINE INPUT #1,Y
1110  NEXT
1120 PRINT VAL(W)-VAL(Y)             'answer
1130 CLOSE
1140 KILL "junk.*"
RUN
 2048
```

The seventh line of output from CHKDSK is the one that says how

many bytes are available on a disk (in DOS 3.3). The first phase above gives us a starting value. The second creates a second small file. The difference in the two reports tells us how many bytes were "allocated" for that second file. A sector is 512 bytes, thus, in this case, a cluster is equal to four sectors. (This was done on a DOS-dedicated 20 MB hard disk.)

Because FAT mechanisms are based on clusters, and because DOS file processes chain from one cluster to another, and because time lags occur between the updating of FAT, DIR, and data areas, chains can become disjointed or intertwined. We that write applications must KNOW when this happens, if we want to guarantee the accuracy of the data we are being paid to watch over.

One technique useful for ascertaining the integrity of file cluster-chaining is to store relative record numbers in the records themselves (as in the MRI-ISAM routines shown above). When we OPEN a file, for example, GET the last record in the file using LOF (divided by the length of one record). Check the number stored in that record. If it is the same as the LOF-expression, the end-to-end integrity of the file is Ok. Probably.

Although there is a minuscule chance that a positive test is the result of coincidence, if the above test fails, something is badly wrong. This is at least one definition of what "occasionally" should mean: Run CHKDSK, now.

And, having stored record numbers in the records themselves, we have provided ourselves with a rudimentary aid for acting on that nebulous piece of advice in the user's manual that says "... you should consider repairing the disk."

Presumably, the sage that said that, and he who said that we need not be concerned with how DOS allocates clusters have never had a payroll system go haywire an hour before the pay checks have to be handed out. Nor have they had to wonder why RUN-program or CHAIN suddenly seems to take longer than it used to.

A program file is a sequential file, essentially. When we are loading, and editing, and saving, it is possible that a program becomes disjointed--one chunk is in one cluster, another may be clear across the disk, and the next somewhere in between. And so on. Which causes a load to take longer than need be.

So, we should indeed be "concerned".

A simple technique for ensuring that ready-to-deliver programs individually span contiguous clusters is to do a LOAD, then a SAVE, to a newly formatted floppy. And while we are at it, we can stack the files on that disk in a prioritized order, based on the most frequently used-first, to enhance directory search times.

To ensure all of that work was not for naught, just before a new application is installed on the target machine, the user should run a compress utility on their system disk to ensure that COPY will write the target files into contiguous clusters. By copying the files in our preferred order, their names will be added to directories in the order we intended.

But we still must be careful. Directories are in clusters too. As names are added, when a cluster is filled, another must be allocated. When copying a series of new files onto a disk, we know for sure, if directory add-on clusters must be used, they are going to be interspersed between real files. So, after all new files have been successfully copied, run your favorite compress utility one more time.

Indeed we should be concerned about how clusters are managed. And not just because we are striving for optimum performance. A broken program-file chain can be disastrous. FRE can be the basis for a technique to ensure the integrity of a program.

Get a FRE(0) report with no program in memory. After a LOAD--not a RUN or CHAIN--see what FRE(0) says, then put a test statement near the front of the program. If that dynamic load-test fails, stop the show. RUN may run into some funny bytes that just happen to look like the tokens for KILL, or CHAIN, or NAME, or SHELL, or CALL, or....

During development FRE(0) varies as we edit lines of a program. Editing (and CLEAR) resets FRE(0) to correspond to the amount of memory consumed by a static program, in the same way a LOAD would. A simple technique for ascertaining what the integrity test factor should be is to CLEAR and PRINT FRE(0). If the test-literal has been precoded as single precision, it will occupy four bytes in the tokenized program text. It still will after we edit it to reflect the now current test-size argument.

If FRE(0) reports 60300 with no program in memory, for example,

and we know our current test argument is 34555, then a "cold start" statement like the one below can be used.

```
IF SGN(60300!-34555!-FRE(0)) THEN .... oops
```

Note: A variation of this technique can also be devised to ensure that no one has been doing unauthorized modifications to your program. At a selected warm-start point, use the then current FRE in an expression that uses the last four digits of your Social Security number as an add-on value, for example, and do a NEW. Any editing change will upset the apple cart on a RUN, and, obliterate all conspicuous evidence of where the self-destruct statement is located. Like any 49-cent padlock, the determined can bust it. If you suspect they have, it will be obvious because they had to tamper with an unknown factor in the test expression.

There are no simple and fast ways for doing length-integrity tests on sequential data files. APPEND-files are especially vulnerable to being scattered around the disk in disjointed clusters. For short files, the risks are minimal, of course. Short in this case meaning not longer than the length of one cluster. (One more reason for wanting to know cluster sizes in given situations.)

For the fainthearted, one of the tricks shown in Chapter 14 is for generating BCC (Block Check Code) hash totals. If you are willing to suffer the performance time required, store a BCC in a file. After an OPEN for RANDOM with a record length of one, serially read each byte and generate a new BCC, then compare it to what is expected. A bad answer can mean bad news.

A slightly less cumbersome technique for doing integrity tests on sequential files can be based on knowledge of what should be contained near the end. For traditional files, for example, we expect the last byte to be a CHR$(26), control-Z code.

First, OPEN a sequential file as a relative file with a record length of one byte. If it is file number one, GET 1,LOF(1) will read the last byte. If it is a control-Z, the file may be Ok. If it is not, it is not one that was closed according to DOS traditions, or the DIR length of the file was not updated correctly, or the FAT has been fried, or a chain has been broken, or.... (PS: This is valid technique for files having less than 32768 bytes, only, viz Chapter 8.)

For those less inclined toward paranoia, but that also want the best performance possible, simply do not use large sequential files, or do piggybacking with APPEND. Use relative files, and a technique such as the one described earlier that puts record numbers in the records themselves. Not only can a test be made after OPEN, it can be done after every GET. If GET and got do not agree, something is rotten in the state of DOS. (Or we have a bug in the program that did the PUT.)

As an overall safeguard, consider doing a CHKDSK in the BAT that starts an application. Send the report to a file. When the first program gets going, open the check-file and check its LOF. By knowing the length of a "clean report", a longer file means CHKDSK reported errors. Then....

Anticipating hardware and operating system errors was the theme above. Another suspect that should worry us even more is the operator. Merely saying that they should do a RESTORE is often not enough. If they continue processing with what we have good reason to believe has been corrupted, and back up that over the top of what may have been a "clean" back-up, no mere miracle will ever untangle it all.

A technique based on using NAME to rename a file is a simple, fast, and nearly foolproof way of detecting abnormal endings. Like, when updating is going on. And we want to make sure the program got all the way to RESET (which purges all buffers, and updates directories).

Nearly any static file can be used as a safety check. On any first attempt to write to a data file, just before PUT is done, NAME "FILESAFE.LOG" AS "FILEOPEN.LOG", for example. As a last act, after RESET, use NAME to change open back to safe. In a menu or gateway process then, attempt to OPEN "FILESAFE.LOG". If that attempt fails, it is probably not safe to go ahead.

NAME in BASIC means RENAME to DOS. It can only be done on files that are not open. No data file access is done. No FAT changes have to be made. Only one (directory) sector has to be updated.

Yes, it is possible that the power could go out between a RESET and a NAME change. So, this technique could cause us to insist that a RESTORE be done needlessly. Better to be safe than sorry, however. And the odds of a failure occurring during the

interval between RESET and NAME are pretty remote, even in the mysterious world of DOS and BASIC. (Mysterious because none of our manuals tell us when, or in what specific order such things are done.)

That safety-check file suggested above, with the dot-LOG name extension, can also in fact be a job-log file. Before a RUN to another program is done, for example, make a log entry of what is about to be done. DATE$, TIME$, and the program-name may be useful information later.

> Note: To make a log file general to an overall system, you may want to call it dot-COM, and put it in the root. This will preclude having to specify a path in every program that makes use of it. (DOS can find EXE, COM, and BAT files in the root of the default drive, no matter what path we are executing from. But, it makes no check to see if in fact a COM or EXE or BAT file are what they are supposed to be.)

An error handler that cannot cope with a problem at hand might also log ERR and ERL with DATE$ and TIME$ and that program's name. A log such as this can be an invaluable aid when it comes to having to figure out what really did or did not happen before things went belly-up. (Few operators can remember what they did, precisely, five minutes after a crash.)

Assuming there is a body available for an autopsy, a job-log can also serve as the basis for a technique to prevent a user from doing a RESTORE from an out-of-sequence set of back-ups. Good DP practices are predicated on father-son schemes, with successive back ups done on different volumes of back-up media, used on a rotating basis. Rue the day when a trusted operator inadvertently does a RESTORE with a back-up made last week, instead of from the one done yesterday.

The last several techniques were for guarding ourselves against system and operator failures. The next one to be dredged up acknowledges a different kind of risk--the risk of software changes in future releases--but, ignores that risk because of the need for speed today: CALL machine-code routines when we are coerced into doing so.

Being reluctant to go native is a meritorious attitude. It is a very good way to get into trouble. Not especially because we are inept or incapable, but because someone is apt to pull the rug out from under us in some future software release. My

opinions on this theme have been exhausted elsewhere. Still, there are times when we must prostitute our principles and get our hands dirty.

In BASIC it is easy to clear the monitor, and we can scroll up by putting the cursor on the bottom of the screen and doing a PRINT. The "normal" way to scroll down, however, is by going to the top, and reprinting all lines, one line farther down. And that is s-l-o-w in any language.

A sometimes acceptable alternative in BASIC is to use BSAVE and BLOAD. That simple concept has been covered elsewhere. It is relatively easy to do, and it is fairly quick when we can make use of VDISK, but a little slower for hard disk (and absolutely lethargic when done to floppies).

When we are really pushed to mimic the fast scrolling that most word processing programs can do, we merely need to mimic their technique: BIOS service calls.

The following subroutine is useful as is for doing high speed screen scrolling. It will shift all lines up or down one line. It was taken from a program that has a window that extends from line five, down through twenty. It should be evident how it can easily be modified to suit any size window. (Remembering that machine language arguments are zero-based offsets.)

> PS: An excellent book on BIOS mechanics is called "System BIOS for IBM PC/XT/AT Computers and Compatibles" published by Phoenix Technologies Ltd. It is from that source that the following "arguments" were derived.

```
1500 'pageV
1510 IF CM THEN E=&H1700 ELSE E=&H700 'clear in blue or black
1520 I=&H100:ON FR GOTO 1530,1540,1550,1560:I=0:GOTO 1560
1530  D=&H700:F=(L-1)*256:GOTO 1570  'ins = from L down to 20
1540  D=&H600:F=(L-1)*256:GOTO 1570  'del = from 20 up to L
1550  D=&H700:F=&H400:GOTO 1570      'up key = scroll down
1560  D=&H600:F=&H400                'dn key = scroll up
1570 MID$(V,1)=MKI$(&HB4+D)  'B4dd  mov ah,d  scroll 6=up/7=down
1580 MID$(V,3)=MKI$(&HB0+I)  'B0ii  mov al,i  i rows (0=clear)
1590 MID$(V,5)=MKI$(&HB7+E)  'B7ee  mov bh,ee attribute (7/23)
1600 MID$(V,7)=MKI$(&HB5+F)  'B5ff  mov ch,f  top row (4-19)
1610 MID$(V,9)=MKI$(&HB1)    'B100  mov cl,00 left col (0-79)
1620 MID$(V,11)=MKI$(&H13B6) 'B613  mov dh,13 bottom row (19)
1630 MID$(V,13)=MKI$(&H4FB2) 'B24F  mov dl,4F right col (0-79)
1640 MID$(V,15)=MKI$(&H10CD) 'CD10  int 10    (video func AH)
```

```
1650 MID$(V,17)=MKI$(&H2CA)  'CA02  ret 2     (2*1 param)
1660 MID$(V,19)=MKI$(0)      '00              (2nd word)
1670 DEF SEG:B=VARPTR(V)+1:B=PEEK(B+1)*256+PEEK(B):CALL B(I)
1680 RETURN
```

To make use of the above subroutine, follow these rules:

CM (Color Main) indicates the background color of the area that will be cleared upon completion. As coded here, any number in CM will clear with blue; zero will use black. (See lines 1510 and 1590.)

FR (Function Request) sets up one of five (1-5) options, via the ON GOTO in line 1520.

1 = Insert a blank line and shift those below down 1 line.
2 = Delete a line by up-shifting those below up 1 line.
3 = Scroll all lines down and blank the top one.
4 = Scroll all lines up and blank the bottom one.
5 = Clear the entire window. (Done by any code save 1-4).

For options 1 and 2, L should equal the current CSRLIN. When scrolling is done, the (now) blank line needs to be repainted with a LOCATE and a PRINT.

Notice that V has already been established as a string (with a DEFSTR) and that it is already at least 20-bytes long, and that all other variables have been prenamed. And that there is another reason for including this handy module in this chapter:

This is a technique for calling any machine language routine. Simply manufacture code in any string variable. And to heck with passing arguments. Force-fit them as literals in the machine code itself, just like an assembler would do it. (Hint: DEBUG is an easy way to see what machine codes look like in assembler language.)

This basic concept can be varied, of course. Routines that have no options can be set up as string constants. Longer routines can be fetched from an external file as prefab sequences with nulls occupying positions that are to be modified with dynamic arguments. If 255 bytes are not enough, use multiple strings initially allocated back-to-back. The key suggestions here are to execute machine code sequences from strings, rather than by shortening the default 64kb BASIC page size. And we can skip fooling around with the passing of parameters. (Which is how the manuals encourage us to do it.)

By the way, CALL in QuickBASIC must be done differently. (Not better, just different.) One more aggravation to contend with when designing programs that we want to use either way. It seems some folks get perverse pleasure out of making our lives more complicated than need be. What they tell us in just one sentence can sentence us unfairly if we fail to remember such gotchas when choosing techniques.

Having seen my ideas for executing machine language subroutines from within a BASIC program, here is another handy one: Ever wish you could write a "COM" program in BASIC? Like the TYPE command, for example, but much improved so that pages can be read without scrolling faster than you can hit Pause. We can easily write our own "general purpose" programs. What is not obvious is how to pass a file name (or switches) to our programs directly from a DOS-prompt command line.

Use a BAT file to start the ball rolling; pass whatever is wanted in BAT-file "variables". Like this:

```
GWBASIC MYTYPE %1 %2 %3
```

That's the easy part. To invoke MYTYPE.BAT, and cause it to open a file named on the DOS command line, as in

```
MYTYPE README.DOC
```

we need to be able to get ahold of "README.DOC", just like the big boys do it. Inside the MYTYPE program, the following start up sequence will do it.

```
1000 'getDTA
1010 DEF SEG:DEFINT C-L:DEFSTR M-Z:B=0:BS=0:BP=0:I=0
1020 MC=STRING$(128,0) 'machine code
1030 B=VARPTR(MC)+1:B=PEEK(B+1)*256+PEEK(B)
1040 MID$(MC,1)=CHR$(&H6)                    'push es
1050 MID$(MC,2)=CHR$(&H53)                   'push bx
1060 MID$(MC,3)=CHR$(&H1E)                   'push ds
1070 MID$(MC,4)=CHR$(&H52)                   'push dx
1080 MID$(MC,5)=MKI$(&H2FB4)                 'mov ah,2f get DTA
1090 MID$(MC,7)=MKI$(&H21CD)                 'int 21h
1100 MID$(MC,9)=CHR$(&H6)                    'push es
1110 MID$(MC,10)=CHR$(&H53)                  'push bx
1120 MID$(MC,11)=CHR$(&H5A)                  'pop dx
```

```
1130 MID$(MC,12)=CHR$(&H1F)                 'pop ds
1140 MID$(MC,13)=MKI$(&H25B4)               'mov ah,25 set int
1150 MID$(MC,15)=MKI$(&H60B0)               'mov al,60   int#
1160 MID$(MC,17)=MKI$(&H21CD)               'int 21h
1170 MID$(MC,19)=CHR$(&H5A)                 'pop dx
1180 MID$(MC,20)=CHR$(&H1F)                 'pop ds
1190 MID$(MC,21)=CHR$(&H5B)                 'pop bx
1200 MID$(MC,22)=CHR$(&H7)                  'pop es
1210 MID$(MC,23)=MKI$(&HCA)                 'ret 0       exit
1220 MID$(MC,25)=MKI$(0)                    'end
1230 CALL B
1240 DEF SEG=0
1250 BS=PEEK(387)*256+PEEK(386)             'PSP def seg address
1260 BP=PEEK(385)*256+PEEK(384)             'DTA offset
1270 DEF SEG=BS
1280  FOR I=1 TO PEEK(BP):MID$(MC,I)=CHR$(PEEK(BP+I)):NEXT
1290  I=INSTR(2,MC," ")+1
1300  OPEN MID$(MC,I,PEEK(BP)-I-1) FOR INPUT AS 1
1310            'la de da, and so on, hereafter
```

This works because: GWBASIC is an EXE program. DOS technical manuals tell us that a PSP--Program Segment Prefix--is set up by the loader routine in COMMAND.COM. This is just as true for the interpreter program as it is for any EXE file. We are also told that a default DTA--Data Transfer Area--is contained in the PSP at offset 80h in the PSP. And that whatever is typed on a command line, after the initiating program's name, will be a string of bytes in the DTA. And that the first byte of the DTA string will be a length-of-string value of from 0 to 127.

So: The above machine language instruction sequence makes a DOS function call to find out where the DTA is for GWBASIC.EXE. Then it saves that gem of wisdom in the Interrupt Vector Table as if it was the address of a "user" interrupt servicing routine, for interrupt 60h. (See Chapter 11 about passing parameters in this sacred area.)

The loop in 1280 copies the string pointed to by our "interrupt" into our program. In this case, it will contain two names. The first one is the name of the program that GWBASIC was told to load. The second name is the one we want. Line 1290 is a crude parser to find the beginning of the second name. PS: Notice that BASIC effectively ignores this name, but it is sitting up in its DTA area just the same.

And see how we can do anything we want to in BASIC. From the operator's perspective, MYTYPE can be made to work as if it was

written in C, Pascal, or any of the other favorites of those that will argue that BASIC is incapable of such sophisticated feats.

Since we have already entered the red light district, we may as well go on and get our money's worth: Programs that modify themselves, or others....

There are at least two rational reasons for using nefarious techniques of this type. It costs time--and sometimes more than a little space--to store things in files for the benefit of subsequent processes. Here is an alternative, practical for several types of problems (strictly in GW-BASIC).

```
OPEN "program.bas" AS 1 LEN = 7 : FIELD 1,7 AS X
```

Chapter 2 describes what a tokenized program file looks like; an abbreviated reminder is sufficient to see the significance of the opening sequence above.

The 7-byte-X record size permits a straight shot into the first line of a program, i.e., record number two. Here is what is skipped over in record number one:

Byte 1: CHR$(255) or CHR$(254), depending on whether SAVE was done with the P-to-protect option.

Word 1: A 2-byte in-memory real address value for the first line of the program.

Word 2: Another 2 bytes; the line number of the first line.

Word 3: Two tokens. If the first line is coded as a DATA statement, these two bytes should be a code 132 and a 32, in decimal terms. (The token for DATA, followed by a space.)

Because we are clever, and because we know DATA statements will always contain text-characters, and because we put no unneeded spaces between the line number and the word DATA, we can jab whatever codes we want to in our pseudo-record number two. Or, three, or four, or more, as long as we "know" how long that first line is, and our READ logic knows what to expect.

One good reason for resorting to this trick is for tailoring a set of programs with an installation program. That program can solicit the answers to a variety of questions, then implant the answers into each of the programs in the run-time set with an

OPEN-PUT-RESET sequence. Those programs then merely need to do a READ into specific variables to see what they want to know. Which is faster and less cumbersome than having to read some application global-file that contains such information every time a program is used.

Other uses can be made of this technique, of course. Such as "remembering" what an operator last did when a program was last used. Just before termination, simply open the program then in use, then save what you want it to remember. The next time it is RUN, READ will provide instant recall without taxing anyone's patience. (Next check-number, or the serial number to be used for the next invoice or purchase order are natural candidates for this technique. It can also be the fundamental basis for doing a did-it-run-till-done test.)

And see how this deviation can still adhere to the adopted rules described in Chapter 11. If the first line is a DATA statement, simply shift the program identification REM-info down onto line 1010, following the GOTO that gets the show on the road. Resorting to sinful techniques (in the eyes of some) does not mean that we should forsake our religious principles altogether.

Lest the obvious be missed, see also how easy it is to get rid of the protect-lock for programs that were saved with comma-P. Simply GET 1,1 and stuff a CHR$(255) in byte-1, then PUT 1,1. It is a mystery to me why the manuals make a mystery out of this. Clever kids can figure this out; if they were not clever they would not be learning to program. Would they?

Few clients seem to care much about what techniques we use to solve their problems. They simply care about results. Until they find they have been trapped into a costly corner, anyway.

Most often our decisions about alternative techniques is a private affair. Still, good or bad, right or wrong, clumsy or clever should be serious matters of debate in our own mind.

What has been exhibited here are not proposed techniques; they are merely meant to be extensions to your own list of possible alternatives. Even if your list is only slightly longer now, neither of us has wasted our time.

To turn off the high order bit of a byte held in an integer variable, use AND. The decimal number to use on one side of AND is 32639. Which is 32767-128. Half of 65536 is 32768. A 2-byte word, 16-bits, can represent a positive whole number of from 0 to 65535. And on, and on, and on.

It is not hard to remember that Boolean operators can be used to flip bits on and off. What is hard for me to remember is which number to use for specific cases, and how to construct the expression. Especially if I have not done it in a month or so. Or a week, or sometimes even a day or two.

It is not hard to figure it out all over again, of course, but it may take a few minutes of experimenting in direct execution mode. And a few more minutes to test it thoroughly. And....

Even if you are a trivia champ with instant recall, a lot that must be done in a lot of programs is way more than trivial. And even if you enjoy thinking out complicated algorithms, not many of us enjoy the labor involved in thoroughly testing a newly reinvented one. Depending on just how complicated a piece of coding is, the time required for thorough testing can be a lot more than just a few minutes.

One alternative is to hunt up an old program that used the same expression. Or, if in a hurry, or feeling lazy, just say to heck-with-it and use IF and THEN and ELSE, or anything else that requires less headwork. Or, copy one from this chapter.

Here are some freebees, loosely grouped by type of function rather than by what kind of programs they are useful in. A binary search is a searching trick usable in payroll programs, shots at the moon, and in games for kids.

And "tricks" some of these are. Some use so-called standard programming practices. Some are so perverse that they should be used only in private. Hours were spent upon the issue of including those that might be viewed as belonging only in a red-light district. The outcome was to include everything from my notebooks that might be usable to someone else.

Judge me not, and I'll do likewise. Sometimes, anything that works ought to be legal. One criteria for judging is the old

saw that a good program is one that works--one that doesn't may be pretty, and pretty useless. Copy anything that follows that you like. Ignore anything that seems obscene. Many of your own tricks may be far superior to mine. No inference is intended that any of these are the "best way" to do something.

My coding style reflects personal habits, one of which must be known up front. With no notes to the contrary assume always:

```
  DEFDBL A     'Accounting Amounts (money) and Accumulations
  DEFSNG B     'Bigger numbers, Binary addresses and BASIC
  DEFINT C-L   'Counters, Do-loops, Indexes and Limits
  DEFSTR M-Z   'Messages, MID$, and the rest of the Zoo
```



```
1080 'find MEAN average, sorted array
1090 '  call:  A(n)= table of numbers (anytype)
1100 '         F= 1st element, L= Last element; max= 32767
1110 '  exit:  A= A(F) + A(L) divided by 2
1120 A=(A(F)+A(L))/2            'answer
```

```
1130 'find MEAN average, unsorted array
1140 '  call:  A(n)= table of numbers (anytype)
1150 '         F= 1st element, L= Last element
1160 '  exit:  A= A(lowval) + A(hival) divided by 2
1170 '  temp:  I= Incr, J= lowval ptr, K= hival ptr
1180 J=L                        'swag lowval ptr
1190 K=L                        'swag hival ptr
1200  FOR I=F TO L
1210   IF A(I)<A(J) THEN J=I    'new lowval ptr
1220   IF A(I)>A(K) THEN K=I    'new hival ptr
```

```
1230  NEXT
1240 A=(A(K)+A(J))/2            'answer


1250 'find MEDIAN average, sorted array
1260 '  call:  A(n)= table of numbers (anytype)
1270 '         F= 1st element, L= Last element
1280 '  exit:  A= median of A(first) and A(last)
1290 '  temp:  I= mid ptr-1 of A(n) if L-F is odd
1300 '         J= mid ptr+1 of A(n) if L-F is odd
1310 I=(L-F)\2+F                'mid ptr rounded up
1320 J=(L-F+1)\2+F              'mid ptr rounded down
1330 A=(A(I)+A(J))/2            'answer


1340 'find MODE (norm), unsorted array
1350 '  call:  T(n)= table (anytype)
1360 '         F= 1st element, L= Last element
1370 '  exit:  G= Got ptr of most-of in T(n)
1380 '            latter-one of dupes
1390 '  temp:  I= Incr, H= Had ptr
1400 '         K= had cnt, J= found cnt, E= exit
1410 FOR J=L TO F STEP-1        'redef L
1420  FOR I=F TO L              'init Found
1430   IF T(I)=T(J) AND I<>J THEN G=I:I=L:L=J:J=0
1440  NEXT
1450 NEXT                       'J starts as -1
1460 FOR E=L TO F STEP-1
1470  K=0                       'reset had cnt
1480  FOR I=F TO L              'sample loop
1490   IF T(I)<>T(G) THEN IF K=0 THEN H=I:K=-1
1500   IF T(I)=T(H) AND K<0 THEN K=K-1
1510   IF J>K THEN I=L:G=H:J=K  'Got replaces Had
1520  NEXT
1530  IF ABS(J)>E THEN E=0      'early finish
1540 NEXT


1550 'find SIMPLE average, unsorted array
1560 '  call:  A(n)= table of numbers (anytype)
1570 '         F= 1st element, L= Last element
1580 '  exit:  A= sum of A(all) divided by L
1590 '  temp:  I= Incr
1600 A=A(F)                     'first
1610  FOR I=F+1 TO L:A=A+A(I):NEXT
1620 A=A/L                      'answer
```

```
1140 'bit EXPRESSION examples
1150 '  call:  E= integer (using low-order byte)
1160 '  mask:  |128|64|32|16|8|4|2|1|
1170 '  bit #  |  7| 6| 5| 4|3|2|1|0|
1180 E=E-32*(E>64 AND E<91)  'force lower case
1190 E=E+32*(E>96 AND E<123) 'force upper case
1200 E=E+32*(E>96 AND E<123)-32*(E>64 AND E<91) 'flip case
1210 E=E AND 32639           'force 7-bit off


1220 'bit RESET (make binary 0)
1230 E=E OR E XOR 8     'set #3 OFF
1240 E=E OR E XOR 68    'set OFF bits #6 and #2 (68= 64+4)


1250 'bit REVERSAL (toggle ON/OFF state)
1260 E=E XOR 32         '#5 REVERSED
1270 E=E XOR 21         '#4, #2, & #0 REVERSED (21= 16+4+1)
```

```
1280 'bit SET (make binary 1)
1290 E=E OR 16          'set ON #4
1300 E=E OR 48          'set ON #5 and #4 (48= 32+16)


1310 'bit SHIFT (all bits, 1 position in unison)
1320 E=E/2              'RIGHT (#0 lost, #7 is 0)
1330 E=E*2 AND 255      'LEFT (#7 lost, #0 is 0)


1340 'bit TEST (sample for binary 0 or 1)
1350 IF E AND 8 THEN    'true for #3 ON
1360 IF E AND 4=0 THEN  'true for #2 OFF
1370 IF E AND 33 THEN   'true for #5 & #0 ON (33= 32+1)


1380 'get GREATER of 2 numbers
1390 '  call:  I= any number, J= any number
1400 '  exit:  E= greater of I,J  (I and J unchanged)
1410 '  note:  logic equals:  IF I>J THEN E=I ELSE E=J
1420 E=I*ABS(I=>J)+J*ABS(J>I)


1430 'get GREATER of 2 strings
1440 '  call:  X= any string, R= any string
1450 '  exit:  S= the greater of X,R  (X and R unchanged)
1460 '  note:  logic equals:  IF X>R THEN S=X ELSE S=R
1470 S=LEFT$(X,LEN(X)*-(X=>R))+LEFT$(R,LEN(R)*-(R>X))


1480 'get SMALLER of 2 numbers
1490 '  call:  I= any number, J= any number
1500 '  exit:  E= smaller of I,J  (I and J unchanged)
1510 '  note:  logic equals:  IF I<J THEN E=I ELSE E=J
1520 E=I*ABS(I<J)+J*ABS(J<=I)


1530 'get SMALLER of 2 strings
1540 '  call:  X= any string, R= any string
1550 '  exit:  S= the smaller of X,R  (X and R unchanged)
1560 '  note:  logic equals:  IF X<R THEN S=X ELSE S=R
1570 S=LEFT$(X,LEN(X)*-(X<R))+LEFT$(R,LEN(R)*-(R<=X))


1580 'display BYTES in numeric variables
1590 '  call:  A= any value, DEFtype A as needed
1600 '  temp:  I= Incr, B= var adrs, G= var type
```

```
1610 '         C= byte, F= Factor bits
1620 G=ASC(VARPTR$(A))-1:B=VARPTR(A)    'var type & adrs
1630 FOR I=0 TO G:C=PEEK(B+I)           'chr loop
1640  IF C<31 THEN PRINT "."; ELSE PRINT CHR$(C);
1650 PRINT SPC(8);:NEXT:PRINT
1660 FOR I=0 TO G:C=PEEK(B+I)           'hex loop
1670  PRINT STRING$(-1*(C<16),48);HEX$(C);SPC(7);
1680 NEXT:PRINT
1690 FOR I=0 TO G:C=PEEK(B+I)           'octal loop
1700  PRINT STRING$(-1*(C<64),48);STRING$(-1*(C<8),48);
1710  PRINT OCT$(C);SPC(6);
1720 NEXT:PRINT
1730 FOR I=0 TO G:C=PEEK(B+I):F=128     'bits loop
1740  WHILE F:PRINT CHR$(48+SGN(C AND F));
1750  F=F\2:WEND:PRINT " ";
1760 NEXT:PRINT
```

```
1170 'convert BCD (Binary Coded Decimal) to DECIMAL
1180 '  call:  X= bytes in range 00H-99H
1190 '  exit:  S= ASCII digits 0-9
1200 '  temp:  I= Incr
1210 S=""
1220 FOR I=1 TO LEN(X)
1230  S=S+HEX$(ASC(MID$(X,I)))
1240 NEXT


1250 'convert BINARY to DECIMAL
1260 '  call:  X= ASCII zeros and ones
1270 '  exit:  A= positive whole number
1280 '  temp:  I= Incr, B= factor position
1290 A=0:B=1
1300 FOR I=LEN(X) TO 1 STEP-1
1310  A=A+(ASC(MID$(X,I)) MOD 2)*B:B=B*2
1320 NEXT


1330 'convert BINARY to HEXADECIMAL
1340 '  call:  X= ASCII zeros and ones
1350 '  exit:  S= ASCII, 0-F, X= length adj MOD 4
1360 '  temp:  I= Incr, J= hex digit
1370 X=STRING$((4-LEN(X) MOD 4)*-(LEN(X) MOD 4>0),48)+X:S=""
1380 FOR I=1 TO LEN(X) STEP 4
1390  J=VAL(MID$(X,I,1))*8+VAL(MID$(X,I+1,1))*4
1400  J=J+VAL(MID$(X,I+2,1))*2+VAL(MID$(X,I+3,1))
1410  S=S+MID$("0123456789ABCDEF",J+1,1)
1420 NEXT


1430 'convert BINARY to OCTAL
1440 '  call:  X= ASCII zeros and ones
1450 '  exit:  S= ASCII 0-7, X= length adj MOD 3
1460 '  temp:  I= Incr, J= octal digit
1470 X=STRING$((3-LEN(X) MOD 3)*-(LEN(X) MOD 3>0),48)+X:S=""
1480 FOR I=1 TO LEN(X) STEP 3
1490  J=VAL(MID$(X,I,1))*4+VAL(MID$(X,I+1,1))*2
1500  J=J+VAL(MID$(X,I+2,1)):S=S+CHR$(48+J)
1510 NEXT


1520 'convert DECIMAL to BCD (Binary Coded Decimal)
```

```
1530 '  call:  X= ASCII digits 0-9
1540 '  exit:  S= bytes, range 00H-99H, X= length adj MOD 2
1550 '  temp:  I= Incr
1560 X=STRING$(LEN(X) MOD 2,48)+X:S=""
1570 FOR I=1 TO LEN(X) STEP 2
1580  S=S+CHR$((ASC(MID$(X,I))-48)*16+ASC(MID$(X,I+1))-48)
1590 NEXT


1600 'convert DECIMAL to BINARY
1610 '  call:  A= positive whole number
1620 '  exit:  S= ASCII zeros and ones, A= 1
1630 '  temp:  I= Incr, AQ= quotient
1640 S="":A=A+1
1650 FOR I=A>1 TO 0:AQ=INT(A/2)
1660  S=CHR$(48-(A=AQ*2))+S:A=A-AQ:I=A>1
1670 NEXT


1680 'convert DECIMAL to HEXADECIMAL
1690 '  call:  A= positive whole number
1700 '  exit:  S= ASCII, 0-F, A= 0
1710 '  temp:  I= Incr, J= hex digit
1720 I=0:WHILE A=>16^I:I=I+1:WEND:S=""
1730 FOR I=I-1 TO 0 STEP-1:J=INT(A/(16^I))
1740  S=S+MID$("0123456789ABCDEF",J+1,1):A=INT(A-(J*16^I))
1750 NEXT


1760 'convert DECIMAL to OCTAL
1770 '  call:  A= positive whole number
1780 '  exit:  S= ASCII, 0-7, A= 0
1790 '  temp:  I= Incr, J= octal digit
1800 I=0:WHILE A=>8^I:I=I+1:WEND:S=""
1810 FOR I=I-1 TO 0 STEP-1:J=INT(A/8^I)
1820  S=S+CHR$(48+J):A=INT(A-(J*8^I))
1830 NEXT


1840 'convert HEXADECIMAL to BINARY
1850 '  call:  X= ASCII 0-F
1860 '  exit:  S= ASCII zeros and ones
1870 '  temp:  I= Incr, Q= translate string
1880 Q="0000000100100011010001010110011110001
       0011010101111001101111011111"
1890 S=""
1900 FOR I=1 TO LEN(X)
```

```
1910 S=S+MID$(Q,(INSTR("123456789ABCDEF",MID$(X,I,1))*4)+1,4)
1920 NEXT


1930 'convert HEXADECIMAL to DECIMAL
1940 '  call:  X= ASCII 0-F
1950 '  exit:  A= positive whole number
1960 '  temp:  I= Incr
1970 A=0
1980 FOR I=LEN(X) TO 1 STEP-1
1990  A=INT(A)+INSTR("123456789ABCDEF",MID$(X,I,1))
2000 A=A*16^(LEN(X)-I):NEXT


2010 'convert HEXADECIMAL to OCTAL
2020 '  call:  X= ASCII 0-F
2030 '  exit:  S= ASCII 0-7
2040 '  temp:  I= Incr, J= hex digit, A= decimal
2050 S="":A=0
2060 FOR I=LEN(X) TO 1 STEP-1
2070  A=INT(A)+INSTR("123456789ABCDEF",MID$(X,I,1))
2080 A=A*16^(LEN(X)-I):NEXT:I=0:WHILE A=>8^I:I=I+1:WEND
2090 FOR I=I-1 TO 0 STEP-1:J=INT(A/8^I)
2100  S=S+CHR$(48+J):A=INT(A-(J*8^I))
2110 NEXT


2120 'convert OCTAL to BINARY
2130 '  call:  X= ASCII 0-7
2140 '  exit:  S= ASCII zeros and ones
2150 '  temp:  I= Incr, Q= translate string
2160 Q="000001010011100101110111":S=""
2170 FOR I=1 TO LEN(X)
2180  S=S+MID$(Q,(INSTR("1234567",MID$(X,I,1))*3)+1,3)
2190 NEXT


2200 'convert OCTAL to DECIMAL
2210 '  call:  X= ASCII 0-7
2220 '  exit:  A= positive whole number
2230 '  temp:  I= Incr
2240 A=0
2250 FOR I=LEN(X) TO 1 STEP-1
2260  A=INT(A)+INSTR("1234567",MID$(X,I,1))*8^(LEN(X)-I)
2270 NEXT
```

```
2280 'convert OCTAL to HEXADECIMAL
2290 '  call:  X= ASCII 0-7
2300 '  exit:  S= ASCII 0-F
2310 '  temp:  I= Incr, J= octal digit, Q= translate string
2320 Q="000001010011100101110111":S=""
2330 FOR I=1 TO LEN(X)
2340  S=S+MID$(Q,(INSTR("1234567",MID$(X,I,1))*3)+1,3)
2350 NEXT
2360 Q=STRING$((4-LEN(S) MOD 4)*-(LEN(S) MOD 4>0),48)+S:S=""
2370 FOR I=1 TO LEN(Q) STEP 4
2380  J=VAL(MID$(Q,I,1))*8+VAL(MID$(Q,I+1,1))*4
2390  J=J+VAL(MID$(Q,I+2,1))*2+VAL(MID$(Q,I+3,1))
2400  S=S+MID$("0123456789ABCDEF",J+1,1)
2410 NEXT
```



```
1060 'generate BCC (Block Check Code) 1-byte hash
1070 '  call:  X= string less than 255 bytes
1080 '  exit:  X= X+CHR$(bcc), as often used in RS232
1090 '  temp:  I= Incr, K= bcc hash
1100 X=LEFT$(X,254)+CHR$(0):K=0                'len(X)<255
1110 FOR I=1 TO LEN(X)-1 STEP 2
1120  K=K XOR CVI(MID$(X,I))                   'pairs
1130 NEXT
1140 K=PEEK(VARPTR(K)) XOR PEEK(VARPTR(K)+1)    'fold over
1150 MID$(X,LEN(X))=CHR$(ABS(K))                'insert BCC


1160 'generate COKE codes (consonants only keys)
1170 '  call:  X= the "name", upper case ASCII, len<255
1180 '  exit:  S= any 1st ltr + consonants, no doubles
1190 '  temp:  I= Incr, C= ptr
1200 S=X+" ":IF MID$(S,2,1)=LEFT$(S,1) THEN MID$(S,2)="."
1210 FOR I=2 TO LEN(S)-1
1220  C=INSTR("BCDFGHJKLMNPQRSTVWXYZ",MID$(S,I,1))
1230  IF C=0 THEN MID$(S,I)="."
1240  IF MID$(S,I,1)=MID$(S,I+1,1) THEN MID$(S,I)="."
```

```
1250 NEXT:C=INSTR(S,".")
1260 WHILE C:MID$(S,C)=MID$(S,C+1):C=INSTR(S,"."):WEND
1270 S=LEFT$(S,INSTR(S," "))


1280 'generate SOUNDEX code (phonetic search key)
1290 '  call:  X= the "name" in upper case ASCII
1300 '  exit:  S= 1st letter of name + 3 ASCII digits
1310 '  temp:  I= Incr, J= scan, C= ptr
1320 S="0000":MID$(S,1,1)=X:C=2
1330 FOR I=2 TO LEN(X)
1340  J=INSTR("RMNLDTCGJKQSXZBFPV",MID$(X,I,1))          'key
1350  IF J THEN MID$(S,C,1)=MID$("65543322222222111",J) 'sub
1360  IF J THEN C=C+1:IF C>4 THEN I=255
1370 NEXT



DATA TRANSLATION ..................................... line

determine CURRENCY denominations (US) ................ 1120

mask-off high order BIT (#7) in character strings .... 1390

shift LOWER case ASCII letters to UPPER case ......... 1460

shift UPPER case ASCII letters to LOWER case ......... 1530

switch UPPER and LOWER case ASCII letters ............ 1600

tokenize repeated CHARACTERS in ASCII strings ........ 1670

token-expand repeated CHARACTERS in ASCII strings .... 1800

translate BYTES of strings using find/swap strings ... 1880

translate ORDINAL number to CARDINAL string .......... 1990


1120 'determine CURRENCY denominations (US)
1130 '  call:  A= positive dollars amount
1140 '         S= string, len>9
1150 '  exit:  monitor output, S= string amount
1160 '  temp:  I= Incr, K= cnt
1170 PRINT USING "#######.##";A;:PRINT STRING$(10,29);:K=1
1180 FOR I=POS(0) TO POS(0)+9
1190  MID$(S,K)=CHR$(SCREEN(CSRLIN,I)):K=K+1:NEXT:PRINT
```

```
1200 K=VAL(LEFT$(S,4))
1210  IF K THEN PRINT K;"thousands";MKI$(-(K=1)*8221)
1220 K=VAL(MID$(S,5,1))
1230  IF K THEN PRINT K;"hundreds";MKI$(-(K=1)*8221)
1240 K=VAL(MID$(S,6,2))
1250  IF K>49 THEN K=K-50:PRINT " 1 fifty"
1260  IF K>39 THEN K=K-40:PRINT " 2 twenties"
1270  IF K>19 THEN K=K-20:PRINT " 1 twenty"
1280  IF K>9  THEN K=K-10:PRINT " 1 ten"
1290  IF K>4  THEN K=K -5:PRINT " 1 five"
1300  IF K THEN PRINT K;"ones";MKI$(-(K=1)*8221)
1310 K=VAL(RIGHT$(S,2))
1320  IF K>74 THEN K=K-75:PRINT " 3 quarters"
1330  IF K>49 THEN K=K-50:PRINT " 2 quarters"
1340  IF K>24 THEN K=K-25:PRINT " 1 quarter"
1350  IF K>19 THEN K=K-20:PRINT " 2 dimes"
1360  IF K>9  THEN K=K-10:PRINT " 1 dime"
1370  IF K>4  THEN K=K -5:PRINT " 1 nickle"
1380  IF K THEN PRINT K;"pennys";MKI$(-(K=1)*8221)


1390 'mask-off high order BIT (#7) in character strings
1400 '  call:  Q= any string
1410 '  exit:  Q= with all bytes < chr$(128)
1420 '  temp:  I= Incr
1430 FOR I=1 TO LEN(Q):C=ASC(MID$(Q,I))
1440  MID$(Q,I)=CHR$(C AND 32639)
1450 NEXT


1460 'shift LOWER case ASCII letters to UPPER case
1470 '  call:  Q= any string
1480 '  exit:  Q= with no lower case
1490 '  temp:  I= Incr, C= chr val
1500 FOR I=1 TO LEN(Q):C=ASC(MID$(Q,I))
1510  MID$(Q,I)=CHR$(C-32*(C>64 AND C<91))
1520 NEXT


1530 'shift UPPER case ASCII letters to LOWER case
1540 '  call:  Q= any string
1550 '  exit:  Q= with no upper case
1560 '  temp:  I= Incr
1570 FOR I=1 TO LEN(Q):C=ASC(MID$(Q,I))
1580  MID$(Q,I)=CHR$(C+32*(C>96 AND C<123))
1590 NEXT
```

```
1600 'switch UPPER and LOWER case ASCII letters
1610 '  call:  Q= any string
1620 '  exit:  Q= with all upper/lower cases reversed
1630 '  temp:  I= Incr
1640 FOR I=1 TO LEN(Q):C=ASC(MID$(Q,I))
1650  MID$(Q,I)=CHR$(C+32*(C>96 AND C<123)-32*(C>64 AND C<91))
1660 NEXT


1670 'tokenize repeated CHARACTERS in ASCII strings
1680 '  call:  X= string, S= pack-character (often " ")
1690 '  exit:  X= repeats tokenized CHR$(127+ # of S)
1700 '            tokens follow S-characters
1710 '  temp:  I= Incr, J= cnt, E= exit
1720 E=LEN(X)
1730 E=E+(E-128)*(E>128)        'max 128 or len(X)
1740 FOR I=E TO 3 STEP-1        'trips at least
1750  J=INSTR(X,STRING$(I,S))   'repititions of S
1760   IF J THEN X=LEFT$(X,J)+CHR$(127+I)+MID$(X,J+I)
1770  I=I-(J>0)                 'same sequence again?
1780   IF INSTR(X,STRING$(3,S))=0 THEN I=3
1790 NEXT


1800 'token-expand repeated CHARACTERS in ASCII strings
1810 '  call:  X= string, bytes > CHR$(128) are tokens
1820 '  exit:  X= token-byte-1 repeated, token-128 times
1830 '  temp:  I= Incr, E= expander
1840 FOR I=LEN(X) TO 2 STEP-1   'scan right-to-left
1850  E=ASC(MID$(X,I))          'token test
1860  IF E>128 THEN X=LEFT$(X,I-1)+
                     STRING$(E-128,MID$(X,I-1))+MID$(X,I+1)
1870 NEXT


1880 'translate BYTES of strings using find/swap strings
1890 '  call:  X= any string
1900 '         Q= find-in-this string
1910 '         S= swap-with-in string
1920 '  exit:  E= len(X) or, 0 if len(S)<>len(Q)
1930 '         X= translated if E, else unchanged
1940 '  temp:  I= Incr, C= ptr
1950 E=LEN(X)*-(LEN(Q)=LEN(S))
1960 FOR I=1 TO E:C=INSTR(Q,MID$(X,I,1))
1970  IF C THEN MID$(X,I)=MID$(S,C,1)
1980 NEXT
```

```
1990 'translate ORDINAL number to CARDINAL string
2000 '  call:  Q= translate string, len=>213
2010 '         X= mask string, len=>9
2020 '         P= parse string, len=>11
2030 '         A= input number <= 99,999,999.99
2040 '  temp:  I= Incr, L= cnt, C= cnt, K= cents
2050 '  exit:  printed, I= L= C= junk, A=unchanged
2060 '  note:  output is akin to "check amounts"
2070 LSET Q="1one2two3three4four5five6six7seven8eight"
2080  MID$(Q,41)="9nine:ten;eleven<twelve=thirteen"
2090  MID$(Q,73)=">fourteen?fifteen@sixteenAseventeen"
2100  MID$(Q,108)="BeighteenCnineteenDtwentyEthirty"
2110  MID$(Q,140)="FfortyGfiftyHsixtyIseventyJeighty"
2120  MID$(Q,173)="KninetyLhundredMthousandNmillionO"
2130 L=CSRLIN:C=POS(0):PRINT USING "########.##";A;:LSET X=""
2140 FOR I=1 TO 11:MID$(X,I)=CHR$(SCREEN(L,C)):C=C+1:NEXT
2150 PRINT:K=VAL(RIGHT$(X,2)):LSET P=LEFT$(X,8)
2160  WHILE ASC(P)=32:LSET P=MID$(P,2):WEND:J=77
2170 FOR I=INSTR(P," ")-3 TO 2 STEP-3
2180  MID$(P,I+1)=MID$(P,I):MID$(P,I)=CHR$(J):J=J+1
2190 NEXT
2200 FOR I=INSTR(P," ")-2 TO 2 STEP-4:C=VAL(MID$(P,I-1,1))
2210  IF C THEN MID$(P,I+1)=MID$(P,I):MID$(P,I)="L"
2220 NEXT:C=1
2230 L=INSTR(P,"N000M"):IF L THEN MID$(P,L+1)=MID$(P,L+5,80)
2240 FOR I=1 TO INSTR(P," ")-1:L=ASC(MID$(P,I))
2250  J=VAL(MID$(P,I,2)):IF J>9 AND J<20 THEN L=J+48:I=I+1
2260  IF J>19 THEN L=J\10+66:IF J MOD 10 THEN J=-1 ELSE I=I+1
2270  E=INSTR(Q,CHR$(L))
2280  IF E THEN LSET X=MID$(Q,E+1,INSTR(E,Q,CHR$(L+1))-E-1)
2290  IF C THEN MID$(X,1)=CHR$(ASC(MID$(X,1))-32)
2300  IF E THEN PRINT LEFT$(X,INSTR(X," ")-1);
2310  C=VAL(MID$(P,I+1,1))*(L>66 AND L<76)
2320  IF C THEN PRINT "-"; ELSE IF L-48 THEN PRINT " ";
2330 C=0:NEXT:IF INT(A)=0 THEN PRINT "Zero ";
2340 PRINT "Dollar";STRING$(ABS(INT(A)<>1),115);
2350 PRINT " and";K;"Cent";STRING$(ABS(K<>1),115)
```

```
1200 'calendar MONTH display, years 1901-2000
1210 '  call:  T= string len=>182, Y= string len=>100
1220 '         I= year (1901-2000), J= month (1-12)
1230 '  temp:  T= months, Y= years, E= exit, I= days
1240 '         K= 1st day, L= line, C= col, H= hold, J= flag
1250 '  exit:  display on monitor
1260     LSET Y ="CDEMABCKFGAIDEFNBCDL" '1901-20
1270 MID$(Y,21) ="GABJEFGHCDEMABCKFGAI" '1921-40
1280 MID$(Y,41) ="DEFNLCDLGABJEFGHCDEM" '1941-60
1290 MID$(Y,61) ="ABCKFGAIDEFNBCDLGABJ" '1961-80
1300 MID$(Y,81) ="EFGHCDEMABCKFGAIDEFN" '1981-00
1310     LSET T ="A144725736146B255136147257"
1320 MID$(T,27) ="C366247251361D477351362472"
1330 MID$(T,53) ="E511462473513F622573514624"
1340 MID$(T,79) ="G733614625735H145136147257"
```

```
1350 MID$(T,105)="I2562472513613J367351362472"
1360 MID$(T,131)="K471462473513L512573514624"
1370 MID$(T,157)="M623614625735N734725736146"
1380 K=VAL(MID$(T,INSTR(T,MID$(Y,I-1900,1))+J,1))
1390 E=ASC(MID$("303232332323",J))-(I MOD 4=0 AND J=2)-20
1400 PRINT " Su Mo Tu We Th Fr Sa":H=1:I=1
1410 FOR L=1 TO 6:FOR C=1 TO 7:J=(H<K OR I>E)
1420  IF J THEN PRINT "   "; ELSE PRINT USING "###";I;
1430 I=I-(J=0):H=H+1:NEXT:PRINT:NEXT


1440 'compute DAY of WEEK for years 1901-2000
1450 '  call:  X= string, Julian date as yyddd, len=>5
1460 '         Q= string, len=>101
1470 '  exit:  B= cvs(3-letter-day-name-space)
1480 '         E= 0 if X is not logical
1490 '  temp:  Q= translate string (day years begin on)
1500 '         I= year, J= day pointer
1510 LSET Q="56123467124567234571235671345":MID$(Q,29)=Q
1520 MID$(Q,4)=LEFT$(Q,101):MID$(Q,1)="734"       'years=Jan 1
1530 I=ABS(VAL(LEFT$(X,2))):E=VAL(MID$(Q,I+1,1))  'year starts
1540 J=((VAL(MID$(X,3,3))+E-1) MOD 7+1)*4         'day pointer
1550 B=CVS(MID$("   Sat Sun Mon Tue Wed Thu Fri ",J))
1560 E=(VAL(MID$(X,3,3))<=365-(I MOD 4=0))        'logic check
1570 IF E THEN PRINT MKS$(B)                      'display


1580 'convert DATE, Gregorian to Julian
1590 '  call:  X= string, mm/dd/yy (assumed valid)
1600 '  exit:  B= single precision whole number, yyddd
1610 '  temp:  I= Incr
1620 B=VAL(MID$(X,4,2))                           'date
1630 FOR I=VAL(LEFT$(X,2))-1 TO 1 STEP-1
1640  B=B+ASC(MID$("CACBCBCCBCBC",I))-36
1650 NEXT                                         'per month
1660 B=B+((B>59)*SGN(VAL(RIGHT$(X,2)) MOD 4))     'leap year
1670 B=B+VAL(RIGHT$(X,2))*1000                    'append year


1680 'convert DATE, Julian to Gregorian
1690 '  call:  B= whole number, yyddd (assumed valid)
1700 '         S= string, len=8
1710 '  exit:  S= mm/dd/yy, B= junk
1720 '  temp:  I= month, J= days
1730 RSET S=STR$(INT(B/1000)+100)                 'get year
1740 B=B-INT(B/1000)*1000                         'get days
1750 FOR I=1 TO 12
```

```
1760  J=ASC(MID$("C@CBCBCCBCBC",I))-36
1770  J=J-(I=2 AND VAL(S) MOD 4=0)            'leap year
1780  IF B<=J THEN MID$(S,3)=STR$(I+100):I=12
1790 B=B-J:NEXT                               'per month
1800 MID$(S,1)=STR$(B+J+100)                  'format
1810 MID$(S,1)=MID$(S,5,2):MID$(S,4)=MID$(S,3,2)
1820 MID$(S,3)="/":MID$(S,6)="/"


1830 'convert TIME, 12-hour (AM/PM) to 24-hour
1840 '  call:  X= hhmmssAM or PM, len=>7
1850 '            assumed valid ("M" not used)
1860 '         R= string, len=>8
1870 '  exit:  R= hh:mm:ss
1880 '  temp:  I= hour, E= noon/midnight
1890 I=VAL(LEFT$(X,2))-12*(LEFT$(X,2)<="12")    'nighttime
1900 E=12*(INSTR(X,"P")=0)                      'morning
1910 E=E+12*(I=24)-12*(LEFT$(X,7)="120000A")
1920 LSET R=STR$(I+E+100):LSET R=MID$(R,3)      'format
1930 MID$(R,4)=MID$(X,3):MID$(R,7)=MID$(X,5)
1940 MID$(R,3)=":":MID$(R,6)=":"


1950 'convert TIME, 24-hour to 12-hour (AM/PM)
1960 '  call:  X= hh:mm:ss, len=>8, assumed valid
1970 '         R= string, len=>7
1980 '  exit:  R= hhmmssdM (d= "A" or "P")
1990 '            "M" included if R is long enough
2000 '  temp:  I= hour, E= 1 or 2 (AM/PM)
2010 I=12*(X>"12:59:59")-12*(VAL(X)=0)+VAL(X)   'adj hour
2020 E=2+(LEFT$(X,8)<"12:00:00" OR VAL(X)=24)   'set AM/PM
2030 LSET R=STR$(I+100):LSET R=MID$(R,3)        'format
2040 MID$(R,3)=MID$(X,4,2):MID$(R,5)=MID$(X,7)
2050 MID$(R,7)=MID$(" AMPM",E*2)


2060 'elapsed DAYS, Julian dates, 1900-1999
2070 '  call:  X= fromdate, yyddd, len=>5 (assumed valid)
2080 '         R= thrudate, yyddd, len=>5 (assumed valid)
2090 '  exit:  B= days elapsed
2100 '         E= 0 if from/thru reversed
2110 '  temp:  I= Incr, J= fromyear, K= thruyear
2120 J=VAL(LEFT$(X,2)):K=VAL(LEFT$(R,2))    'from:thru
2130 B=0                                    'clear
2140 FOR I=J TO K-1
2150  B=B+365-(I MOD 4=0)
2160 NEXT                                   'per year
```

```
2170 B=B+VAL(MID$(R,3,3))-VAL(MID$(X,3,3))  'subtract
2180 E=(B=>0)*(LEFT$(R,5)>LEFT$(X,5))       'logical?


2190 'elapsed TIME, 12-hour, hhmmssA or hhmmssP
2200 '  call:  S= start time (assumed valid), len=>7
2210 '         X= end time (assumed valid), len=>7
2220 '         R= string, len=>6
2230 '  exit:  R= elapsed time (hhmmss)
2240 '  temp:  I= hours, J= minutes, K= seconds, E= flag
2250 K=VAL(MID$(X,5,2))-VAL(MID$(S,5,2))        'seconds
2260 J=VAL(MID$(X,3,2))-VAL(MID$(S,3,2))+(K<0)  'minutes
2270 I=VAL(LEFT$(X,2))-VAL(LEFT$(S,2))+(J<0)    'hours
2280 E=(RIGHT$(S,1)<>RIGHT$(X,1))               'AM/PM flag
2290 K=K-60*(K<0):J=J-60*(J<0)                  'adjust
2300 I=I-12*(I<0)-12*(I<0 AND E=0)-12*(I=>0 AND E<0)
2310 LSET R=STR$(I+100):I=CVI(MID$(R,3))        'format
2320 LSET R=STR$(J+100):J=CVI(MID$(R,3))
2330 RSET R=STR$(K+100):MID$(R,3)=MKI$(J):MID$(R,1)=MKI$(I)


2340 'elapsed TIME, 24-hour, hh:mm:ss
2350 '  call:  S= start time (assumed valid), len=>8
2360 '         X= end time (assumed valid), len=>8
2370 '         R= string, len=>8
2380 '  exit:  R= elapsed time (hh:mm:ss)
2390 '  temp:  I= hours, J= minutes, K= seconds
2400 K=VAL(MID$(X,7))-VAL(MID$(S,7))            'seconds
2410 J=VAL(MID$(X,4))-VAL(MID$(S,4))+(K<0)      'minutes
2420 I=VAL(X)-VAL(S)+(J<0)                      'hours
2430 K=K-60*(K<0):J=J-60*(J<0):I=I-24*(I<0)     'adjust
2440 LSET R=STR$(I+100):I=CVI(MID$(R,3))        'format
2450 LSET R=STR$(J+100):J=CVI(MID$(R,3))
2460 RSET R=STR$(K+100):MID$(R,4)=MKI$(J):MID$(R,1)=MKI$(I)
2470 MID$(R,3)=":":MID$(R,6)=":"


2480 'fielded DATE, Julian, 2-bytes, encode/decode
2490 '  call:  B= yyddd (assumed valid)
2500 '         R= 2-byte string (typically fielded)
2510 '  exit:  B= decoded R, R= encoded B
2520 '  temp:  C= year, D= days
2530 C=B/1000:D=B-C*1000                    'encode
2540  IF D>255 THEN C=C+128:D=D-128
2550   LSET R=CHR$(C):MID$(R,2)=CHR$(D)

2560 C=ASC(R):D=ASC(MID$(R,2))              'decode
```

```
2570  IF C>127 THEN C=C-128:D=D+128
2580   B=C*1000+D


2590 'reformat DATE, ddmmmyy as mm/dd/yy
2600 '  call:  X= ddmmmyy (assumed valid), len=>7
2610 '         R= string, len=>8
2620 '  exit:  R= mm/dd/yy
2630 '  temp:  J= month number
2640 J=INSTR(" ANEBARPRAYUNULUGEPCTOVEC",MID$(X,4,2))\2
2650 LSET R=STR$(J+100):LSET R=MID$(R,3)
2660 MID$(R,4)=MID$(X,1):MID$(R,7)=MID$(X,6)
2670 MID$(R,3)="/":MID$(R,6)="/"


2680 'reformat DATE, mm/dd/yy as ddmmmyy
2690 '  call:  X= mm/dd/yy (assumed valid), len=>8
2700 '         R= string, len=>7
2710 '  exit:  R= ddmmmyy
2720 '  temp:  none
2730 LSET R=MID$(X,4,2):MID$(R,6)=MID$(X,7,2)
2740 MID$(R,3)=MID$("JFMAMJJASOND",VAL(X),1)
2750 MID$(R,4)=MID$("AEAPAUUUECOE",VAL(X),1)
2760 MID$(R,5)=MID$("NBRRYNLGPTVC",VAL(X),1)


2770 'reformat DATE, mm/dd/yy as month day, year
2780 '  call:  X= mm/dd/yy (assumed valid), len=>8
2790 '         R= string, len=>19
2800 '  exit:  R= month-name daySS, year
2810 '         (SS= st,nd,rd or th)
2820 '  temp:  I= Instr, J= month
2830 J=VAL(X):I=(J MOD 3+1)*9-8
2840             LSET R=MID$("March    January  February",I,9)
2850 IF J>3 THEN LSET R=MID$("June     April    May",I,9)
2860 IF J>6 THEN LSET R=MID$("SeptemberJuly     August",I,9)
2870 IF J>9 THEN LSET R=MID$("December October  November",I,9)
2880 I=INSTR(R," "):MID$(R,I)=STR$(VAL(MID$(X,4)))
2890        I=INSTR("   1  21 31 2  22 3  23",MID$(R,I,3))+1
2900 MID$(R,INSTR(R,"  "))=MID$("th,st,st,st,nd,nd,rd,rd,",I,3)
2910 MID$(R,INSTR(R,",")+1)=STR$(1900+VAL(MID$(X,7)))


2920 'validate DATE, Gregorian
2930 '  call:  X= mm/dd/yy, len=>8
2940 '  exit:  E= 0 if X is invalid
2950 '  temp:  none
```

```
2960 E=32-VAL(MID$(" 141212112121",VAL(LEFT$(X,2))+1,1))
2970 E=E-(E=28 AND (VAL(MID$(X,7,2)) MOD 4=0))
2980 E=E*VAL(MID$(X,4,2))*(VAL(MID$(X,4,2))<=E)
2990 E=E*SGN(VAL(X))*(VAL(LEFT$(X,2))<13)
3000 E=E*(MID$(X,3,1)=MID$(X,6,1))*(MID$(X,3,1)="/")
3010 E=E*SGN(INSTR("01",MID$(X,1,1)))*(LEN(X)>7)
3020 E=E*SGN(INSTR("0123456789",MID$(X,2,1)))
3030 E=E*SGN(INSTR("0123",MID$(X,4,1)))
3040 E=E*SGN(INSTR("0123456789",MID$(X,5,1)))
3050 E=E*SGN(INSTR("0123456789",MID$(X,7,1)))
3060 E=E*SGN(INSTR("0123456789",MID$(X,8,1)))


3070 'validate DATE, Julian
3080 '  call:  X= yyddd, len=>5
3090 '  exit:  E= 0 if X is invalid
3100 '  temp:  none
3110 E=(VAL(LEFT$(X,2)) MOD 4=0)
3120 E=VAL(MID$(X,3,3))*(VAL(MID$(X,3,3))<=365-E)
3130 E=E*SGN(INSTR("0123456789",MID$(X,1,1)))
3140 E=E*SGN(INSTR("0123456789",MID$(X,2,1)))
3150 E=E*SGN(INSTR("0123",MID$(X,3,1)))*(LEN(X)>4)
3160 E=E*SGN(INSTR("0123456789",MID$(X,4,1)))
3170 E=E*SGN(INSTR("0123456789",MID$(X,5,1)))


3180 'validate TIME, 12-hour, hhmmssA or hhmmssP
3190 '  call:  X= hhmmssyb, len=>7
3200 '         (y is A or P, b is blank, null, or M)
3210 '  exit:  E= 0 if X is invalid
3220 '  temp:  none
3230 E=(LEN(X)=7 OR INSTR("M ",MID$(X,8,1))<>0)
3240 E=E*(VAL(LEFT$(X,2))<13)*SGN(VAL(LEFT$(X,2)))
3250 E=E*INSTR("01",MID$(X,1,1))*INSTR("AP",MID$(X,7,1))
3260 E=E*INSTR("0123456789",MID$(X,2,1))
3270 E=E*INSTR("012345",MID$(X,3,1))
3280 E=E*INSTR("0123456789",MID$(X,4,1))
3290 E=E*INSTR("012345",MID$(X,5,1))
3300 E=E*INSTR("0123456789",MID$(X,6,1))


3310 'validate TIME, 24-hour, hh:mm:ss
3320 '  call:  X= hh:mm:ss, len=>8
3330 '  exit:  E= 0 if X is invalid
3340 '  temp:  none
3350 E=(LEFT$(X,2)<>"24" OR LEFT$(X,8)="24:00:00")
3360 E=E*(VAL(LEFT$(X,2))<25)*INSTR("012",MID$(X,1,1))
```

```
3370 E=E*SGN(INSTR("0123456789",MID$(X,2,1)))
3380 E=E*INSTR("012345",MID$(X,4,1))
3390 E=E*SGN(INSTR("0123456789",MID$(X,5,1)))
3400 E=E*INSTR("012345",MID$(X,7,1))
3410 E=E*INSTR("0123456789",MID$(X,8,1))
3420 E=E*(MID$(X,3,1)=MID$(X,6,1))*(MID$(X,3,1)=":")
3430 E=E*(LEFT$(X,8)<>"00:00:00")
```



```
1080 'edit DOLLARS, floating-$, $ZZZ,ZZZ,ZZD.DD-
1090 '  call:  A= whole number, S= string, len=>16
1100 '  exit:  S= edited string, right justified
1110 '  temp:  I= Instr, L= len(S)
1120 LSET S=STR$(INT(A)/100):MID$(S,1)="-"
1130 I=INSTR(S,"  "):L=LEN(S)
1140  IF INSTR(S,".")=0 THEN MID$(S,I)=".00":I=I+3
1150  IF I-INSTR(S,".")=2 THEN MID$(S,I)="0":I=I+1
1160  IF LEFT$(S,2)="-." THEN MID$(S,2)=LEFT$(S,L)
1170  IF LEFT$(S,2)="--" THEN MID$(S,1)="-0":I=I+1
1180 RSET S=LEFT$(S,I):MID$(S,L)=CHR$(32-13*SGN(A<0))
1190  I=L+7*(VAL(LEFT$(S,L-7))<0)
1200  IF I<L THEN MID$(S,1)=MID$(S,2,I):MID$(S,I)=","
1210  I=L+11*(VAL(LEFT$(S,L-11))<0)
1220  IF I<L THEN MID$(S,1)=MID$(S,2,I):MID$(S,I)=","
1230  MID$(S,INSTR(S,"-"))="$"


1240 'edit DOLLARS, floating-(, (ZZZ,ZZZ,ZZD.DD)
1250 '  call:  A= whole number, S= string, len=>16
1260 '  exit:  S= edited string, right justified
1270 '  temp:  I= Instr, L= len(S)
1280 LSET S=STR$(INT(A)/100):MID$(S,1)="-"
1290 I=INSTR(S,"  "):L=LEN(S)
```

```
1300  IF INSTR(S,".")=0 THEN MID$(S,I)=".00":I=I+3
1310  IF I-INSTR(S,".")=2 THEN MID$(S,I)="0":I=I+1
1320  IF LEFT$(S,2)="-." THEN MID$(S,2)=LEFT$(S,L)
1330  IF LEFT$(S,2)="--" THEN MID$(S,1)="-0":I=I+1
1340 RSET S=LEFT$(S,I):MID$(S,L)=CHR$(32-9*SGN(A<0))
1350  I=L+7*(VAL(LEFT$(S,L-7))<0)
1360  IF I<L THEN MID$(S,1)=MID$(S,2,I):MID$(S,I)=","
1370  I=L+11*(VAL(LEFT$(S,L-11))<0)
1380  IF I<L THEN MID$(S,1)=MID$(S,2,I):MID$(S,I)=","
1390  MID$(S,INSTR(S,"-"))=CHR$(32-8*SGN(A<0))


1400 'edit PHONE number as (999) 999-9999
1410 '  call:  A= 10-digit whole number, S= string len=>14
1420 '  exit:  S= (zzz) zzz-zzzz, left justified, zero filled
1430 '  temp:  I= inspect for spaces
1440 LSET S=STR$(A)
1450  WHILE MID$(S,14,1)=" ":MID$(S,2)=LEFT$(S,13):WEND
1460 MID$(S,2)=MID$(S,5,3):MID$(S,7)=MID$(S,8,3)
1470 MID$(S,5)=")":MID$(S,1)="(":MID$(S,10)="-":I=INSTR(S," ")
1480  WHILE I*(I<15):MID$(S,I)="0":I=INSTR(S," "):WEND
1490 MID$(S,6)=" "


1500 'edit PHONE number as 999-999-9999
1510 '  call:  A= 10-digit whole number, S= string len=>12
1520 '  exit:  S= zzz-zzz-zzzz, left justified, zero filled
1530 '  temp:  I= inspect for spaces
1540 LSET S=STR$(A)
1550  WHILE MID$(S,12,1)=" ":MID$(S,2)=LEFT$(S,11):WEND
1560 MID$(S,1)=MID$(S,3,3):MID$(S,5)=MID$(S,6,3)
1570 MID$(S,4)="-":MID$(S,8)="-":I=INSTR(S," ")
1580  WHILE I*(I<13):MID$(S,I)="0":I=INSTR(S," "):WEND


1590 'edit SOCIAL SECURITY number as 999-99-9999
1600 '  call:  A= 9-digit whole number, S= string len=>11
1610 '  exit:  S= zzz-zz-zzzz, left justified, zero filled
1620 '  temp:  I= inspect for spaces
1630 LSET S=STR$(A)
1640  WHILE MID$(S,11,1)=" ":MID$(S,2)=LEFT$(S,11):WEND
1650 MID$(S,1)=MID$(S,3,3):MID$(S,5)=MID$(S,6,2)
1660 MID$(S,4)="-":MID$(S,7)="-":I=INSTR(S," ")
1670  WHILE I*(I<12):MID$(S,I)="0":I=INSTR(S," "):WEND
```

```
1070 'search for BYTE, largest within a string
1080 '  call:  X= any string
1090 '  exit:  C= asc(largest byte), E= 1st position
1100 '  temp:  I= Incr
1110 C=0
1120 FOR I=1 TO LEN(X)
1130  E=ASC(MID$(X,I)):C=E*ABS(E=>C)+C*ABS(C>E):NEXT
1140 E=INSTR(X,CHR$(C))


1150 'search for BYTE, smallest within a string
1160 '  call:  X= any string
1170 '  exit:  C= asc(smallest byte), E= 1st position
1180 '  temp:  I= Incr
1190 C=-255*(LEN(X)>0)
1200 FOR I=1 TO LEN(X)
1210  E=ASC(MID$(X,I)):C=E*ABS(E<C)+C*ABS(C<=E):NEXT
1220 E=INSTR(X,CHR$(C))


1230 'search for ELEMENT in an array (binary search)
1240 '  call:  F= find, A(n)= array, sorted, ascending
1250 '         H= highest element, L= lowest element
1260 '  exit:  I= position, E= 0 if F is not found
1270 '  temp:  H= High, L= Low, I= Incr, E= Exit
1280 '  note:  for descending order switch less/greater signs
1290 I=H\2:H=H+1:L=L-1
1300 FOR E=0 TO 1
1310  IF F<A(I) THEN H=I:I=I-(H-L)\2
1320  IF F>A(I) THEN L=I:I=I+(H-L)\2
1330 E=ABS(F=A(I) OR I=H OR I=L):NEXT:E=(F=A(I))


1340 'search for SUBSTRING (longest repeated, in a string)
1350 '  call:  X= any string, len>2
1360 '  exit:  F= From (1st one), L=Len, as in mid$(X,F,L)
```

```
1370 '  temp:  I= Instr
1380 '  note:  includes overlaps ("aaaaa" is F= 1, L= 4)
1390 I=LEN(X):L=SGN(I):F=1
1400 WHILE I
1410  I=INSTR(F+1,X,MID$(X,F,L+1)):L=L+SGN(I)
1420  IF I=0 THEN I=INSTR(F+L,X,MID$(X,F+1,L)):IF I THEN F=I
1430  IF I=0 THEN I=INSTR(F+L,X,MID$(X,F,L)):IF I THEN F=I
1440 WEND:F=INSTR(X,MID$(X,F,L))
1450 L=L*-(L>1)      'L=0 if no repeats of at least 2-bytes
```



```
1140 'reverse NAMES, Doe, John J. Jr. as John J. Doe, Jr.
1150 '  call:  X= last, first middle rank
1160 '         S= string, len=>len(X)
1170 '  exit:  S= first middle last, rank
1180 '  temp:  I= ptr, J= ptr
1190 LSET S=X
1200 FOR I=1 TO LEN(S):J=ASC(MID$(S,I))
1210  MID$(S,I)=CHR$(J-32*(J>64 AND J<91)):NEXT
1220   J=INSTR(S," iv")+INSTR(S," ii")
1230   J=J+INSTR(S," jr")+INSTR(S," sr")
```

```
1240   J=J*SGN(INSTR(". ",MID$(S,J+3,1)) OR J+3=LEN(S))
1250   I=INSTR(S," iii")
1260   I=I*SGN(INSTR(". ",MID$(S,I+4,1)) OR I+4=LEN(S))
1270   J=I*ABS(I=>J)+J*ABS(J>I):I=J:IF J=0 THEN J=LEN(S)+1
1280  WHILE I AND I<LEN(S):MID$(S,I)=" ":I=I+1:WEND
1290  I=INSTR(S,","):LSET S=MID$(S,I+2)
1300  MID$(S,INSTR(S,"    ")+1)=LEFT$(X,I-SGN(I)+1)
1310 I=LEN(S):J=I*ABS(I<J)+J*ABS(J<=I):MID$(S,J)=MID$(X,J)


1320 'reverse NAMES, John J. Doe, Jr. as Doe, John J. Jr.
1330 '  call:  X= first middle last, rank
1340 '         S= string, len=>len(X)
1350 '  exit:  S= last, first middle rank
1360 '  temp:  I= ptr, J= ptr
1370 LSET S=X:I=INSTR(S,","):IF I=0 THEN I=LEN(S)
1380 FOR I=LEN(S) TO I STEP-1:MID$(S,I)=" ":NEXT
1390 I=LEN(S):WHILE I>1 AND MID$(S,I,1)=" ":I=I-1:WEND
1400      J=I:WHILE J>1 AND MID$(S,J,1)>" ":J=J-1:WEND
1410      J=J-(MID$(S,J,1)=" ")
1420 LSET S=MID$(S,J):I=INSTR(S,"  "):MID$(S,I)=","
1430 MID$(S,I+2)=LEFT$(X,J-1):I=INSTR(X,", ")
1440 IF I THEN MID$(S,INSTR(S,"  "))=MID$(X,I+1)


1450 'reverse sequence of BYTES in a string
1460 '  call:  X= any string
1470 '  exit:  X= byte sequence reversed
1480 '  temp:  I= Incr, C= Chr
1490 FOR I=1 TO LEN(X)\2:C=ASC(MID$(X,LEN(X)-I+1))
1500  MID$(X,LEN(X)-I+1)=MID$(X,I,1):MID$(X,I)=CHR$(C)
1510 NEXT


1520 'reverse sequence of ELEMENTS in an array
1530 '  call:  T(n)= array, F= 1st position, E= last position
1540 '  exit:  T(n)= element sequence reversed
1550 '  temp:  I= Incr
1560 FOR I=F TO E/2
1570  SWAP T(I),T(E-I)
1580 NEXT


1590 'shuffle significant ELEMENTS to top of an array
1600 '  call:  T(n)= array, F= 1st position, L= last position
1610 '  exit:  T(n)= nulls shifted to "bottom" of table
1620 '  temp:  E= Exit, I= Incr
```

```
1630 '  note:  for numeric array change LEN to SGN
1640 FOR E=F TO L
1650  IF LEN(T(E))=0 THEN I=E ELSE I=L+1
1660  FOR I=L TO I STEP-1
1670   IF LEN(T(I)) THEN SWAP T(E),T(I)
1680  NEXT
1690 NEXT


1700 'sort BYTES of a string, ascending
1710 '  call:  X= any string
1720 '  exit:  X= bytes sorted left-to-right
1730 '  temp:  E= Exit, I= Incr, J= Juggle, L= len(X)
1740 L=LEN(X)
1750 FOR E=L>0 TO 0
1760  FOR I=1 TO L-1:J=MID$(X,I,1)>MID$(X,I+1,1)
1770   IF J THEN MID$(X,I)=MID$(X,I+1,1)+MID$(X,I,1):L=I
1780  NEXT
1790  E=L<I AND L>0
1800 NEXT


1810 'sort BYTES of a string, descending
1820 '  call:  X= any string
1830 '  exit:  X= bytes sorted right-to-left
1840 '  temp:  E= Exit, I= Incr, J= Juggle, L= len(X)
1850 L=LEN(X)
1860 FOR E=L>0 TO 0
1870  FOR I=1 TO L-1:J=MID$(X,I+1,1)>MID$(X,I,1)
1880   IF J THEN MID$(X,I)=MID$(X,I+1,1)+MID$(X,I,1):L=I
1890  NEXT
1900  E=L<I AND L>0
1910 NEXT


1920 'sort ELEMENTS of an array, ascending (bubble-sort)
1930 '  call:  A(n)= array, F= 1st position, L= last position
1940 '  exit:  A(n)= sorted, ascending, positions F thru L
1950 '  temp:  E= Exit, I= Incr
1960 FOR E=-1 TO 0
1970  FOR I=F TO L-1
1980   IF A(I)>A(I+1) THEN SWAP A(I),A(I+1):L=I
1990  NEXT
2000  E=L<I
2010 NEXT
```

```
2020 'sort ELEMENTS of an array, ascending (shell-sort)
2030 '  call:  A(n)= array, F= 1st position, L= last position
2040 '  exit:  A(n)= sorted, ascending, positions F thru L
2050 '  temp:  E= Exit, H= Half, I= Incr, J= Juggle
2060 H=(L-F)/2
2070 WHILE H
2080  FOR I=F TO H+F:E=1
2090   WHILE E:E=0
2100    FOR J=I TO L-H STEP H
2110     IF A(J)>A(J+H) THEN SWAP A(J),A(J+H):E=1
2120    NEXT
2130   WEND
2140  NEXT:H=H\2
2150 WEND


2160 'sort ELEMENTS of an array, descending (bubble-sort)
2170 '  call:  A(n)= array, F= 1st position L= last position
2180 '  exit:  A(n)= sorted, descending, positions F thru L
2190 '  temp:  E= Exit, I= Incr
2200 FOR E=-1 TO 0
2210  FOR I=F TO L-1
2220   IF A(I+1)>A(I) THEN SWAP A(I),A(I+1):L=I
2230  NEXT
2240  E=L<I
2250 NEXT


2260 'sort ELEMENTS of an array, descending (shell-sort)
2270 '  call:  A(n)= array, F= 1st position, L= last position
2280 '  exit:  A(n)= sorted, descending, positions F thru L
2290 '  temp:  E= Exit, H= Half, I= Incr, J= Juggle
2300 H=(L-F)/2
2310 WHILE H
2320  FOR I=F TO H+F:E=1
2330   WHILE E:E=0
2340    FOR J=I TO L-H STEP H
2350     IF A(J+H)>A(J) THEN SWAP A(J),A(J+H):E=1
2360    NEXT
2370   WEND
2380  NEXT:H=H\2
2390 WEND
```

Chapter 15 = TOOLS

Back when Altair and Albuquerque were unusual names to some, and apples were just fruit to everyone, and RENUM had yet to be invented, some of us spent as many hours changing line numbers, sometimes, as we did writing useful code. It was not long, naturally, before we wrote a program that would renumber the lines of another program, all automatically.

We all did it. We programmers. And every programmer had his own little toolbox in which he kept all of his homemade tools. Me too.

Times have changed. My Altair vamoosed years ago. Now we see Washington and New York addresses on the backs of manuals. And not all apples grow on trees. And RENUM is a built-in feature. But most of us still have toolboxes. Me too.

The tools in this chapter are some of my favorites. Favored because they are so necessary (not because they are my own). The need for this minimum set has not diminished. In fact, they are needed more today than ever.

Time was, when 4 KB was a big program, and you left out remarks to conserve memory. Now we can afford 4 KB-worth of remarks, alone. But bigger programs can also make it a bigger job to keep track of what is what, and where what is. A job that can be ever so much easier with just a few tools.

These programs are shared with a proviso: They were handmade by me, for me. They work just fine, on my machine, with my programs. Reiterated differently: They were not written to suit the world. They may not do everything for everyone, on all machines, on all versions of the interpreter, forever and ever. Ad infinitum.

This is not to say they cannot be made to work, differently, or in a different environment. They are all written in BASIC, and they can be overhauled to whatever extent need be. If they fail to work as described, customize them. Hopefully enough help has been provided that you will be able to easily add them to your own toolbox, for your own use.

Compatibility is, undoubtedly, the most overworked word in

computer advertising today. These tools are sensitive. They can cohabit, but not always blissfully. They all depend on knowing certain addresses up in the interpreter's own working storage areas. The addresses used were found in manuals. Not all of them are in any one manual, however, and they are not easily found. But, because they have been published, at least once, somewhere, they are likely to remain unchanged. For a while, anyway. Here are the ones taken advantage of:

```
Using DEF FNB(B)=PEEK(B+1)*256+PEEK(B) then....

FNB(46)   = line number of line currently executing
FNB(48)   = beginning of first line of object program
FNB(856)  = address of first simple variable
FNB(858)  = address of first array variable
FNB(860)  = beginning of free-space (end of variables)
```

The accuracy of the last three addresses is easily enough confirmed by use of VARPTR. They have all been correct, for me, for three successive releases of, three different versions of BASIC interpreters: One with a last name of EXE, and two that are COM files that link-up with that part of the software that is frozen in ROM. If yours is different than mine you may have to do some peeking and poking. In the manuals and in the software.

Another type of generation gap is possible. All of these tools analyze a program in situ. Chapter 2 tells how to examine programs sitting in memory. New gadgets are added to the language from time to time. If the key word tokens or other bits in yours do not align with mine, getting these programs to work correctly may require some research of the type suggested there.

An overview: All of these programs are "mergeable modules". They are stored as files using SAVE with the comma-A option. Obviously their names can be changed to protect the innocent. (And the guilty.) As you can see, they are all numbered beginning with line 9000 so that they can be merged onto the Tail end of an application program. (In my world no regular program has line numbers that even get close to 9000.) All of these tools can be renumbered so as to start with a higher number, if you like. With the exception of VLIST, they must all be numbered high enough to cause them to be situated beyond the last line of a program to be worked on. By numbering them all alike, they can MERGE over the top of each other.

They are all terse. And cryptic, and tricky. But they are also small and fast. They were kept small so that they can be left in a program while it is in development, at a minimum cost to that program's need for memory. The tricks used to make them as efficient as possible are to save me time. Not others. They are not intended to be models of how to write good programs. They are tools. 'nough said. Grab one.

LXREF = Line Numbers Cross Reference

Runs through a program, top to bottom, and compiles a list of all line-references. (GOTO, etc.) Prints a listing, in line-number order, of all lines that are referenced by statements in other lines. Each target-line number is followed by a listing of the line numbers that point to it.

```
Usage:  LOAD "program"     'the object program to be analyzed
        MERGE "LXREF"      'must be last block of code
        RUN 9000           'printer on? on-line? paper?
        Ok                 'ends with an END
        DELETE 9000-       'if no longer needed
```

Rules: Load size is 800 bytes; needs about 8100 more to run. Assumes object program has no invalid line references. (Report will include bad references, same as valid.) Can report up to 999 references; crashes (ERR=9) if too many. Can be RUN repeatedly, once in residence.

```
9000 PRINT "Lxref":DEF SEG:DEFINT I-J:B=PEEK(47)*256+PEEK(46)
9010 H=VARPTR(#1)+51:POKE H,6:FOR I=H+1 TO H+252:POKE I,1:NEXT
9020 POKE I,0:POKE I+1,0:POKE I+2,0:POKE H+11,2:POKE H+12,2
9030 POKE H+14,3:POKE H+15,0:POKE H+132,6:POKE H+143,6
9040 POKE H+28,2:POKE H+29,4:POKE H+31,8:POKE H+34,5:I=0:J=0
9050 F=B:B=PEEK(49)*256+PEEK(48):C=B:A=B:DIM B(999),A(999)
9060 A=PEEK(B+1)*256+PEEK(B):B=B+3:C=PEEK(B)*256+PEEK(B-1)
9070 IF C<F THEN PRINT C;:LOCATE ,1 ELSE 9170
9080 B=B+1
9090 ON PEEK(H+PEEK(B)) GOTO 9080,9110,9140,9110,9120,9130,,9110
9100 B=B+2:GOTO 9090
9110 B=B+PEEK(H+PEEK(B))+1:GOTO 9090
9120 B=B+1:IF PEEK(B)=34 THEN 9080 ELSE IF PEEK(B) THEN 9120
9130 B=A:GOTO 9060
9140 A(I)=C:B(I)=PEEK(B+2)*256+PEEK(B+1):FOR J=I TO 1 STEP-1
9150 IF B(J)<B(J-1) THEN SWAP B(J),B(J-1):SWAP A(J),A(J-1) ELSE
J=0
9160 NEXT:I=I+1:B=B+3:GOTO 9090
9170 H=I-1:FOR I=0 TO H:IF A(I)<0 THEN 9200 ELSE LPRINT B(I),
9180 FOR J=I TO H:IF B(I)=B(J) THEN LPRINT A(J);:A(J)=-1
9190 NEXT:LPRINT:H=H+(A(H)<0)
9200 NEXT:LPRINT:LPRINT DATE$,TIME$:END
```

```
LHITS = Line Numbers Cross Reference (Selective)

        Is like LXREF, save it only lists lines that have
        been addressed since a RUN was last done.

Usage:  LOAD "program"      'the object program to be analyzed
        MERGE "LHITS"       'must be last block of code
        RUN                 'start your program
        BREAK               'keyboard or END or STOP (optional)
        GOTO 9000           'or GOSUB; outputs to printer
        Ok                  'add your own END or add a
        DELETE 9000-        'RETURN to use as a subroutine

Rules:  Load size is 807 bytes; needs about 8100 more to run.
        Can report up to 999 references; crashes (ERR=9) if
        too many.  Can be RUN repeatedly, once in residence.

9000 PRINT "Lhits":DEF SEG:DEFINT I-J:B=PEEK(47)*256+PEEK(46)
9010 H=VARPTR(#1)+51:POKE H,6:FOR I=H+1 TO H+252:POKE I,1:NEXT
9020 POKE I,0:POKE I+1,0:POKE I+2,0:POKE H+11,2:POKE H+12,2
9030 POKE H+13,3:POKE H+14,2:POKE H+15,0:POKE H+132,6:POKE H+143,6
9040 POKE H+28,2:POKE H+29,4:POKE H+31,8:POKE H+34,5:I=0:J=0
9050 F=B:B=PEEK(49)*256+PEEK(48):C=B:A=B:DIM B(999),A(999)
9060 A=PEEK(B+1)*256+PEEK(B):B=B+3:C=PEEK(B)*256+PEEK(B-1)
9070 IF C<F THEN PRINT C;:LOCATE ,1 ELSE 9170
9080 B=B+1
9090 ON PEEK(H+PEEK(B)) GOTO 9080,9110,9140,9110,9120,9130,,9110
9100 B=B+2:GOTO 9090
9110 B=B+PEEK(H+PEEK(B))+1:GOTO 9090
9120 B=B+1:IF PEEK(B)=34 THEN 9080 ELSE IF PEEK(B) THEN 9120
9130 B=A:GOTO 9060
9140 A(I)=C:B(I)=PEEK(B+2)*256+PEEK(B+1):FOR J=I TO 1 STEP-1
9150 IF B(J)<B(J-1) THEN SWAP B(J),B(J-1):SWAP A(J),A(J-1) ELSE
J=0
9160 NEXT:I=I+1:B=B+3:GOTO 9090
9170 H=I-1:FOR I=0 TO H:IF A(I)<0 THEN 9210 ELSE A=B(I)+3
9180 PRINT PEEK(A+1)*256+PEEK(A),
9190 FOR J=I TO H:IF B(I)=B(J) THEN PRINT A(J);:A(J)=-1
9200 NEXT:PRINT:H=H+(A(H)<0)
9210 NEXT:PRINT:PRINT DATE$,TIME$:END
```

VFIND = Variables Finder

Asks you for a variable name to search for, then runs through a program, top to bottom, and displays the line numbers that that name was found in.

```
Usage:  LOAD "program"     'the object program to be analyzed
        MERGE "VFIND"      'must be last block of code
        RUN 9000           'start tool
        VFind?             'enter search argument
        Ok                 'ends with an END
        DELETE 9000-       'if no longer needed
```

Rules:
- Load size is 981 bytes; needs about 450 more to run.
- Will only find names that obey syntax rules.
- Will also find key words that are not tokenized.
- Will report the "B" and "BF" used in graphics (like in LINE), as if they were variable names.
- Include "FN" in front of user defined function names.
- For array names, include the left-parenthesis, only.
- Alpha characters may be upper or lower case.
- Can be RUN repeatedly, once in residence.

```
9000 PRINT "Vfind? ";:DEF SEG:B=PEEK(47)*256+PEEK(46)
9010 DEFSTR M-Z:DEFINT I:H=VARPTR(#1)+51:POKE H,6
9020 FOR I=H+1 TO H+252:POKE I,1:NEXT:POKE I,0:POKE I+1,0
9030 POKE I+2,0:FOR I=H+11 TO H+14:POKE I,2:NEXT
9040 POKE I,0:POKE H+132,6:POKE H+143,6:POKE H+28,2
9050 POKE H+29,4:POKE H+31,8:POKE H+34,5:POKE H+209,9
9060 FOR I=65 TO 90:POKE H+I,3:NEXT
9070 Z=SPACE$(255):F=B:B=PEEK(49)*256+PEEK(48):C=0:A=B
9080 LINE INPUT Q:IF Q>"" THEN PRINT ELSE END
9090 FOR I=1 TO LEN(Q):IF ASC(MID$(Q,I))<97 THEN 9110
9100 MID$(Q,I)=CHR$(ASC(MID$(Q,I))-32)
9110 NEXT:M="######"+STRING$(6,29)
9120 A=PEEK(B+1)*256+PEEK(B):B=B+3:C=PEEK(B)*256+PEEK(B-1)
9130 IF C<F THEN PRINT USING M;C; ELSE PRINT SPC(6):END
9140 B=B+1
9150 ON PEEK(H+PEEK(B)) GOTO
9140,9170,9210,9170,9180,9190,,9170,9200
9160 B=B+2:GOTO 9150
9170 B=B+PEEK(H+PEEK(B))+1:GOTO 9150
9180 B=B+1:IF PEEK(B)=34 THEN 9140 ELSE IF PEEK(B) THEN 9180
9190 B=A:GOTO 9120
9200 I=3:LSET Z="FN":B=B+1:GOTO 9220
9210 I=2:LSET Z=CHR$(PEEK(B)):B=B+1
9220 WHILE
INSTR("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789$#%!.",CHR$(PEEK(B)))
9230 MID$(Z,I)=CHR$(PEEK(B)):B=B+1:I=I+1:WEND
9240 IF PEEK(B)=40 THEN MID$(Z,I)="(":B=B+1:I=I+1
9250 IF LEFT$(Z,I-1)<>Q THEN 9150
9260 PRINT STRING$(6,28);:B=A:GOTO 9120
```

VLIST = Variables Lister

Displays the names of variables presently in use, in the order in which they are stacked in memory, in the variables-storage work area.

```
Usage:  LOAD "program"     'the object program to be analyzed
        MERGE "VLIST"      'anywhere, but not over your lines
        RUN                'start your program
        BREAK              'keyboard or END or STOP (optional)
        GOTO 9000          'or GOSUB; outputs to monitor
        Ok                 'add your own END or add a
        DELETE 9000-       'RETURN to use as a subroutine
```

Rules: Load size is 688 bytes; needs about 54 more to run. Assumes you have no array called O!(report will include this name. Does a default DEF SEG; otherwise has no impact on a live program.

```
9000 DEF SEG:O!(0)=PEEK(857)*256+PEEK(856) 'Vlist
9010 WHILE O!(0)<PEEK(859)*256+PEEK(858):O!(1)=PEEK(O!(0))
9020 O!(0)=O!(0)+2:IF PEEK(O!(0)-1)>127 THEN PRINT "FN";
9030 PRINT CHR$(PEEK(O!(0)-1) AND 32639);
9040 PRINT STRING$(SGN(PEEK(O!(0))),PEEK(O!(0)));
9050 O!(0)=O!(0)+1:O!(2)=PEEK(O!(0))
9060 WHILE O!(2):O!(0)=O!(0)+1:O!(2)=O!(2)-1
9070 PRINT CHR$(PEEK(O!(0)) AND 32639);:WEND:O!(0)=O!(0)+1
9080 PRINT MID$(" %$!...#",O!(1),1):O!(0)=O!(0)+O!(1):WEND
9090 WHILE O!(0)<PEEK(861)*256+PEEK(860):O!(1)=PEEK(O!(0))
9100 O!(0)=O!(0)+2:PRINT CHR$(PEEK(O!(0)-1));
9110 PRINT STRING$(SGN(PEEK(O!(0))),PEEK(O!(0)));
9120 O!(0)=O!(0)+1:O!(2)=PEEK(O!(0))
9130 WHILE O!(2):O!(0)=O!(0)+1:O!(2)=O!(2)-1
9140 PRINT CHR$(PEEK(O!(0)) AND 32639);:WEND
9150 O!(0)=O!(0)+1:PRINT MID$(" %$!...#",O!(1),1);"("
9160 O!(0)=O!(0)+2+(PEEK(O!(0)+1)*256+PEEK(O!(0))):WEND
```

VXREF = Variables Cross Reference

Runs through a program, top to bottom, and finds all of the variable names that are used in each line. Prints each different name found, in alphabetical order. Each name is followed by a list of each of the line numbers that that name was found in.

Usage:
```
LOAD "program"     'the object program to be analyzed
MERGE "VXREF"      'must be last block of code
RUN 9000           'printer on? on-line? paper?
Ok                 'ends with an END
DELETE 9000-       'if no longer needed
```

Rules: Load size is 1563 bytes; needs about 3500 more to run, plus 3 bytes for each named variable; ERR=7 if not enough free space to complete. Maximum number of unique variables is 255; crashes (ERR=9) if too many. Reports "B" and "BF", as used in LINE, as variables. Can be RUN repeatedly, once in residence.

```
9000 PRINT "Vxref":DEF SEG:DEFINT I-J:B=PEEK(47)*256+PEEK(46)
9010 H=VARPTR(#1)+51:POKE H,6:FOR I=H+1 TO H+252:POKE I,1:NEXT
9020 POKE I,0:POKE I+1,0:POKE I+2,0:POKE H+209,9
9030 FOR I=H+11 TO H+14:POKE I,2:NEXT:POKE I,0
9040 FOR J=H+65 TO H+90:POKE J,3:NEXT:POKE H+132,6:POKE H+143,6
9050 POKE H+28,2:POKE H+29,4:POKE H+31,8:POKE H+34,5:J=255
9060 DEFSTR X-Z:Z=SPACE$(255):Y=Z:K=VARPTR(Y)+1
9070 K=PEEK(K+1)*256+PEEK(K)-4:D=B:B=PEEK(49)*256+PEEK(48):C=B
9080 DIM X(J),H(J),A(J):E=VARPTR(A(J)):E=E-(E<0)*65536!+8:A(0)=D
9090 J=0:H(0)=PEEK(B+1)*256+PEEK(B):B=B+3:C=PEEK(B)*256+PEEK(B-1)
9100 IF C<A(0) THEN PRINT C;:LOCATE ,1:C=B:G=E ELSE 9320
9110 B=B+1
9120 ON PEEK(H+PEEK(B)) GOTO
9110,9160,9180,9160,9140,9150,,9160,9170
9130 B=B+2:GOTO 9120
9140 B=B+1:IF PEEK(B)=34 THEN 9110 ELSE IF PEEK(B) THEN 9140
9150 B=H(0):GOTO 9090
9160 B=B+PEEK(H+PEEK(B))+1:GOTO 9120
9170 LSET Z="FN":B=B+1:I=3:GOTO 9190
9180 LSET Z=CHR$(PEEK(B)):B=B+1:I=2
9190 WHILE
INSTR("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789$!#%.",CHR$(PEEK(B)))
9200 MID$(Z,I)=CHR$(PEEK(B)):I=I+1:B=B+1:WEND
9210 IF PEEK(B)=40 THEN MID$(Z,I)="(":I=I+1:B=B+1:GOTO 9230
9220 IF INSTR("AS     ALL    APPEND BASE   OUTPUT SEG    ",
       LEFT$(Z,7)) THEN 9120
9230 IF X(J)=LEFT$(Z,I) THEN 9120
9240 J=INSTR(Y,LEFT$(Z,1)):IF J=0 THEN 9290
9250 IF X(J)=LEFT$(Z,I) THEN D=E-3 ELSE 9280
9260 IF PEEK(D)=J THEN 9120
9270 IF D>G THEN D=D-3:GOTO 9260 ELSE 9300
9280 J=INSTR(J+1,Y,LEFT$(Z,1)):IF J THEN 9250
9290 J=INSTR(Y," "):X(J)=LEFT$(Z,I):H(J)=E:MID$(Y,J,1)=Z
9300 A(J)=E:POKE E,J:POKE E+1,PEEK(C-1):IF E=K THEN ERROR 7
9310 POKE E+2,PEEK(C):E=E+3:GOTO 9120
9320 D=INSTR(Y," ")-1:FOR I=1 TO
D:MID$(X(I),LEN(X(I)))=CHR$(I):NEXT
9330 I=D/2:WHILE I:FOR H=1 TO I:B=1:WHILE B:B=0
9340 FOR J=H TO D-I STEP I:IF X(J)<=X(J+I) THEN 9360
9350 B=J+I:SWAP X(J),X(B):SWAP H(J),H(B):SWAP A(J),A(B)
9360 NEXT:WEND:NEXT:I=I\2:WEND
9370 FOR H=1 TO D:LPRINT LEFT$(X(H),LEN(X(H))-1);" ";
9380 I=ASC(RIGHT$(X(H),1)):FOR B=H(H) TO A(H) STEP 3
9390 IF PEEK(B)=I THEN LPRINT PEEK(B+2)*256+PEEK(B+1);
9400 NEXT:LPRINT:NEXT:LPRINT D,DATE$,TIME$:END
                    ---=== THE END ===---
```