QuickTALK

Crescent Software Volume 2 Free

INSIDE:

- The Future of Languages
  by Greg Lobdell, Microsoft Corp.
- Random File Techniques
- Peeking at QuickBASIC

Table of Contents

QuickTALK is published whenever we get around to it by Crescent Software.
Volume 2, Number 1 Entire contents copyright (c) 1988 by Crescent Software

Written by Ethan Winer
Contributing Writers: Sureta Escobar, Jay Munro, Greg Lobdell
Design, photography, cartoons and whatever: Jay Munro

Crescent Software
11 Grandview Ave.
Stamford, CT 06905
203-846-2500

A message from the publisher

Welcome to the second issue of QuickTALK–Crescent Software's magazine for BASIC programmers.

QuickTALK is more than just a catalog of our products–it includes feature articles, tips, and useful programming techniques. We hope that QuickTALK will also serve as an entertaining way to get to know us.

In only two short years, Crescent Software has become the undisputed leader in programming tools for QuickBASIC and Turbo Basic. We attribute this success to quality products and a genuine concern for our customers and their needs. They have come to depend on us for sound technical advice that extends far beyond merely supporting our products. We gladly provide assistance to hundreds of BASIC programmers each week, and are proud to have the "best attitude" and level of support in the business.

One example of that attutide is this magazine–it is free to anyone who inquires about our products. Another is our willingness to share our knowledge and expertise. Where other companies refuse to provide source code at any cost, we include it free with all our products. Further, the QuickPak Professional manual contains tutorials on BASIC and assembler programming–beyond what is needed to use the package.

IN THIS ISSUE

The past two years have seen a tremendous increase in the popularity of the BASIC language. With the release of QuickBASIC 4 and BASCOM 6, there is no disputing that BASIC is now *the* most powerful language available for personal computers. Microsoft has also made clear their intention to continue improving and enhancing BASIC.

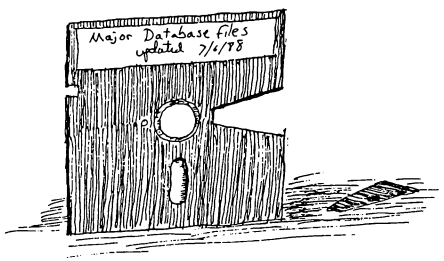
In this issue Microsoft's Greg Lobdell discusses the future of BASIC, and as you will see, many exciting additions to the language are just around the corner. This issue of QuickTALK also includes two tutorials that you're sure to find useful and enlightening.

The first discusses techniques for reading and writing data files, and is intended for beginning to intermediate level programmers. It begins with an overview of file basics, and progresses to a complete discussion of sequential and random access file handling.

For advanced programmers (or those who would like to be), the article "Which Code is Faster?" will really open your eyes! Besides showing you how to write programs that are smaller and run faster, many details of QuickBASIC's internal workings are revealed. You will also see first hand how to use Microsoft's excellent CodeView debugger.

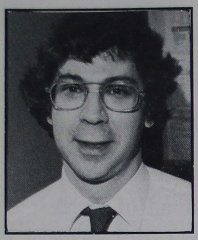
All of us at Crescent Software are excited and pleased to be such an integral part of the growth and popularity of BASIC.

–Ethan Winer



"Sector not found"

Who's Who...

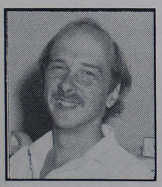


ETHAN WINER founded Crescent Software in 1986 after serving as a consultant to several major corporations. He is a self-taught expert in BASIC, assembly language, and DOS. When not writing programs or conducting business, Ethan writes feature articles for PC Magazine, Programmer's Journal, Borland's Turbo Technix, and several other publications.

Prior to a programming career, he performed professionally as a musician (guitar and electric bass). Ethan is also credited with designing and building one of Connecticut's major recording studios.

SURETA ESCOBAR joined the Crescent team in 1987. Regardless of what Ethan might like to think, it is Sureta who *really* runs the show. An indispensable part of the business, she maintains the daily work flow, coordinates dealer sales and edits and co-writes the Crescent advertising and promotional literature.

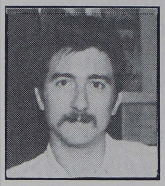
When not assisting customers or playing with a computer, her talents take a different turn. Sureta teaches classes in both Astrology and Parapsychology, and does readings professionally.



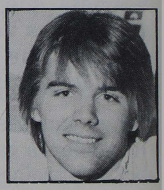
DON MALIN is the brains behind Crescent's QBase relational database, and is responsible for the "look and feel" of all our products. Don came to us one day with an idea and a demo, and the rest is history.

Besides being the best BASIC programmer we know, he is also a Lotus 1-2-3 expert. Don has written and markets a collection of templates and BASIC programs for owners of offshore racing yachts. He was also part of the team that built the racing boat *Freedom,* which won the America's Cup competition.

JAY MUNRO wrote QuickMenu, and serves as Crescent's resident photographer and cartoonist. (Doesn't *every* company need a resident cartoonist?) He's also our favorite person. Also a self taught BASIC and assembly programmer, Jay came to us from Ashton Tate, and has a degree in cinematography.



PAUL PASSARELLI is the head of Tech Support, and provides stability and balance to what would otherwise be total havoc. He posesses a true knack for anything mathematical, and has contributed extensively to QuickPak Professional. Paul is an expert in both BASIC and assembly language, and actually *likes* writing recursive programs.



BRIAN GIEDT is the creator of GraphPak and GraphPak Professional, and he's one of the most amazing guys around. You'll need exponential notation to count the number of languages he knows. Brian has extensive AI experience, he's taught college level courses in programming, and has also developed a powerful hypertext word processor.

RON HOLCOMBE is the author of LaserPak and LaserPak Professional, and comes to us with some very interesting credentials. Programming since 1965, Ron has a B.S. from the U.S. Naval Academy, a BCE and an MSCE in Civil Engineering from Georgia Tech, and served in Vietnam in the Seabees. A mild mannered kind of guy (with a black belt in TaeKwon-do!), he is an award winning amateur musician and songwriter living in Colorado.

Using Random Files in BASIC

by Ethan Winer

If you were to ask most BASIC programmers which area of programming was the most difficult for them to master, the answer would undoubtedly be creating and accessing random data files. While BASIC has always enjoyed a reputation for being the easiest of the high-level languages to learn, there is no disputing that the commands for manipulating database files are often less than obvious. Indeed, this is one of the topics we are most frequently asked to explain. Our focus here will therefore be on the variety of techniques that are used to create, read, and write disk files with fixed length records. Unlike simple sequential files that are accessed with INPUT and PRINT statements, the fixed-length files used in most databases require additional preparation.

FILE BASICS

Before you can read or write any disk file in BASIC, you must first use the OPEN command. There are actually two different forms of OPEN–one being an abbreviated version–but we will use the more formal syntax here because it is clearer. A valid DOS file name must be given, as well as a number which will be used for all subsequent references to the file. You may choose any number you'd like, so long as only one file with that number is open at once.

```
OPEN "Test.Dat" FOR OUTPUT AS #1          'open the file
FOR X = 1 TO 25
    PRINT #1, "This is message number" X  'print the messages
NEXT
CLOSE #1                                  'close the file
```

Figure 1
Sequential files are written using the familiar PRINT command.

The example shown in Figure 1 opens a file named "TEST.DAT" for sequential output, assigns it the number 1, and writes a twenty-five line test message. Notice that once the name and number have been originally specified, only the number is needed when printing to the file.

Because this file is being written to with PRINT statements, a carriage return and line feed will be added at the end of each line automatically by BASIC. This corresponds exactly to the way the PRINT command works when writing to the screen. As you might imagine, sequential files are read using a disk form of the INPUT statement, and the carriage return/line feed pair that was originally added by PRINT tells BASIC when the end of each line has been reached.

```
OPEN "Test.Dat" FOR INPUT AS #1          'open the file
FOR X = 1 TO 25
LINE INPUT #1, Test$                    'read each line
PRINT Test$                             'show it on screen
NEXT
CLOSE #1                                  'close the file
```

Figure 2.

As you can see, I recommend using LINE INPUT rather than INPUT, because INPUT cannot digest quotes, commas, or colons. In this example, we know that twenty-five lines are in the file, so a simple FOR/NEXT loop can be used to read each line. But what should we do if the number of lines to be read is unknown? Attempting to read beyond the end of a sequential file produces an error, and the best way around this problem is to use the BASIC EOF (End Of File) statement just prior to reading each line, as shown below.

```
OPEN "Test.Dat" FOR INPUT AS #1          'open the file
WHILE NOT EOF(1)                         'until the end
    LINE INPUT #1, Test$                 'get a line
    PRINT Test$                          'print it on screen
WEND
CLOSE #1                                 'close the file
```

This is the preferred approach for reading sequential files, since it can accommodate any number of lines without ever causing an error. But there are two fundamental problems with sequential files–BASIC is relatively slow in reading them, and the only practical way to get to the last line is by grinding through all of the lines before it. True, QuickBASIC 4.0 and Turbo Basic offer a binary mode to read any portion of a file, but binary access is not really intended for getting individual lines of text. Further, how would you know where in the file the last line begins?

ENTER RANDOM ACCESS

When a sequential file is being read with INPUT or LINE INPUT, BASIC must examine every single byte as it comes from the disk looking for either a carriage return which marks the end of the line, or a CHR$(26) which marks the end of the file. This takes a considerable amount of time when many lines are to be read. Worse, for each string that is being read, additional time is needed to locate a suitable place in memory to store it. Then, the string data must be placed there, and finally that portion of memory must be marked as being in use.

TIP

There may be times when you want to disable the PrtSc key in a BASIC program. One approach would be to write a TSR that intercepts Interrupt 5. A much easier way is to set the Print Screen *busy flag:*

```
DEF SEG = 0
POKE &H500, 1
```

Then to re-enable it again later:

```
DEF SEG = 0
POKE &H500, 0
```

Random access files are instead read (and written) in pieces, which can quickly be placed into an area of memory that has previously been set aside just for this purpose. Instead of checking each byte as it is being read, BASIC simply grabs a portion of the file in one operation. And because random access files are comprised of fixed length records, it is easy to determine where in the file a given record begins.

Of course, you don't have to calculate the location of the bytes to be read or written—BASIC does this automatically. By specifying the length of each record when the file is first opened, any record can be accessed by simply using its number. This is where the term "random" comes from, since *any* record can be accessed in the file, without having to start at the very beginning. Another feature of random files is that once they have been opened, they can be both read and written at will.

The example below opens a file named "Junk.Dat" for random access, and specifies that each record will have a length of fifty-six bytes.

```
OPEN "Junk.Dat" FOR RANDOM AS #1 LEN = 56
```

When this statement is executed in a running program, a 56 byte area of memory will be set aside as a storage buffer to hold the records to be read or written. As a matter of interest, file buffers are always located in BASIC's string data area. Again, the file buffer is simply a temporary holding area for data on its way to or from a disk file.

OXYMORON?

An oxymoron is a combination of contradictory words. For example, "pretty ugly", "postal service" or "plastic silverware". Comedian George Carlin likes "Jumbo Shrimp" and "Military Intelligence". Our favorite is "RS-232 standard".

Besides telling BASIC the name of the file and the length of each record, you must also define the variables that will be assigned to the file buffer. This is done with the FIELD statement, as shown below.

```
FIELD #1, 25 AS CustName$,    _
          14 AS Phone$,       _
           5 AS ZipCode$,     _
           4 AS Amount$,      _
           4 AS BalanceDue$,  _
           2 AS Account$,     _
           2 AS PriceCode$
```

Once these field assignments have been made, the original buffer space will be divided into separate strings, like those shown in Figure 4. The key point here is that the fifty-six byte buffer memory is set aside when the file is opened, and the positioning of the strings within the buffer is determined by the FIELD statements.

Notice how the underscore character is used to represent a single statement that extends over several program lines. While this certainly increases readability, it is not allowed under QuickBASIC 4. Also notice that even though some of the variables being defined are in fact numeric amounts, they must be represented here as strings.

Let's take a closer look.

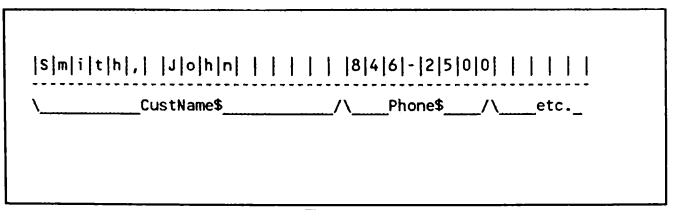


Figure 4

VARIABLE STORAGE

Whenever you assign a string variable, BASIC must locate a free area of memory sufficiently large to hold all of the characters. Of course, short strings occupy less memory than long strings, and the real point is that the amount of memory required varies, depending on the string's length.

Numeric variables are handled very differently, however, and use a fixed number of bytes no matter what value the variable happens to be holding. For example, all double precision numeric variables occupy eight bytes of memory, while a regular integer variable requires only two.

Whenever you enter a numeric variable in response to an INPUT command, BASIC must convert the digits you typed into the appropriate internal format, which takes time. This is yet another contributor to the slowness of BASIC when reading or writing sequential files, because numeric amounts are stored on disk as ASCII digits.

But in a random file, the amounts are instead kept in BASIC's internal format, which allows each record to be the same length. This also means that numbers read in from a random file can be processed very quickly, because the conversion from ASCII digits to a binary value was performed when the record was originally written.

We're not going to belabor the various methods used to represent floating point numbers, because that really isn't important here. What is important are the commands you will use to convert numeric values to strings prior to storing them.

TIP
Some EGA and VGA displays lose the cursor in 43 line mode. This will get it back every time:

```
DEF SEG = 0
POKE &H487, PEEK(&H487) OR 1
```

Four different commands are provided to convert each of the numeric variable types to string form, and another four will convert strings back to numbers. MKI$ (Make an Integer into a String) takes the two bytes that represent an integer variable and converts them into string form. For example, the integer value 288 is stored internally as two bytes—32 and 1—which is illustrated below.

```
X% = 288
Address = VARPTR(X%)
DEF SEG = VARSEG(X%)
PRINT PEEK(Address), PEEK(Address + 1)
PRINT PEEK(Address) + 256 * PEEK(Address + 1)
```

This results in the numbers 32 and 1 being displayed, followed by the total combined value 288. What MKI$ does is to locate the variable as we have, and create a string from the individual bytes. Both of the program fragments below do exactly the same thing, though as you can see, MKI$ is much simpler to use.

```
Value$ = MKI$(X%)
Value$ = CHR$(PEEK(Address)) + CHR$(PEEK(Address + 1))
```

The method used to store single and double precision numbers is more complicated, and as I mentioned before we won't bother with the exact formulas. To create a string variable from a single precision number requires the MKS$ function, while MKD$ does the same thing for a double precision amount. The last converting function is MKL$, and it is used to convert BASIC's long integers to a string. Once the values have been converted to string form, they may then be assigned to the variables that were declared as part of the field statement.

In the example above, Value$ was assigned directly using the MKI$ function. Unfortunately, normal string assignments cannot be used when filling the variables in a field buffer, because of the way BASIC allocates string memory. Each time a string is assigned–even if it was already defined earlier–a new area of memory is set aside to hold it. But since the variables kept in a field buffer must stay in the same place, we can't just use the normal assignment statements. Remember, one of the reasons random files are so fast is because their buffer memory location is fixed when the file was first opened.

LSET, RSET, and MID$

BASIC provides three commands that will let you assign a string that already exists without changing its location–LSET, RSET, and the statement form of MID$. All of these work by letting you replace characters within a string, as opposed to creating an entirely new string. Let's take a closer look at each of these statements in turn.

If you are designing a database program to keep track of, say, the names and addresses of your customers, at some point you must decide how many characters are to be allowed for each field. I usually limit address fields to thirty-two characters, because that's how many can comfortably fit on a standard 3-1/2 inch wide mailing label.

Using that as an example, then, suppose an address happens to occupy only twenty characters. What you'd really like is to place the name into the first twenty character positions in the string, and then pad the remaining twelve places with blanks. This is precisely what LSET does. It is important to understand that if the trailing positions in a field are *not* blanked out, then any remnants left over from a prior read or write will still be present. Again, the same area of memory is used repeatedly for all file reads and writes.

RSET is similar to LSET, except it *right* justifies the new string into the existing field variable. With either LSET or RSET, attempting to insert a string that is too large assigns only as many characters as will fit, without creating an error. Notice that besides their intended use for assigning field variables, LSET and RSET can also be used to advantage in many other programming situations. Since new space isn't established each time the string is assigned, these commands can work very quickly, while minimizing the "clutter" that normally occurs in the string data area. Most programmers use MID$ to extract a portion of a string, but it can also be used to to insert characters, much like LSET and RSET. But where LSET and RSET fill any unused character positions with blanks, MID$ instead leaves them undisturbed.

The syntax for using MID$ to assign characters is the same as when it is used to extract characters–you specify the starting position in the string, as well as the number of characters to include. Like LSET and RSET, if you attempt to replace too many characters in a string with MID$, those that don't fit will be omitted without causing an error. By the way, the MID$ length parameter is optional, and if it is omitted, all of the characters through the end of the string will be included.

PUTTING IT ALL TOGETHER

Now that we've seen the individual steps needed to prepare a field buffer, let's put it all together into a single program. The example in Figure 5 first opens the file and defines all of the string variables that comprise the field buffer. Next, each variable is assigned a value, and then the information is written to a disk record. The last step reads the *next* record from the file, and re-assigns its contents to the original variables for display.

TIP
Some programmers like to make a file untypeable by embedding a CHR$(26) near the beginning. Here's how you can view it:

```
Copy filename /b con:
```

```
CName$     = "Passarelli, Paul"     'make up some data
Phon$      = "(203)-846-2500"
Zip$       = "12345"
Amt!       = 102.45
BalDue!    = 398.77
Acct%      = 158
PrcCode%   = 32

RecordNumber = 123

OPEN "Stuff.Dat" FOR RANDOM AS #1 LEN = 56   'open the file
FIELD #1, 25 AS CustName$,     _              'set up fields
          14 AS Phone$,        _
           5 AS ZipCode$,      _
           4 AS Amount$,       _
           4 AS BalanceDue$,   _
           2 AS Account$,      _
           2 AS PriceCode$

LSET CustName$     = CName$          'assign field variables
LSET Phone$        = Phon$
LSET ZipCode$      = Zip$
LSET Amount$       = MKS$(Amt!)
LSET BalanceDue$   = MKS$(BalDue!)
LSET Account$      = MKI$(Acct%)
LSET PriceCode$    = MKI$(PrcCode%)

PUT #1, RecordNumber                 'write to record #123
RecordNumber = RecordNumber + 1      'point to next record
GET #1, RecordNumber                 'read it from disk

PRINT CustName$
PRINT Phone$
PRINT ZipCode$
PRINT CVS(Amount$)
PRINT CVS(BalanceDue$)
PRINT CVI(Account$)
PRINT CVI(PriceCode$)

CLOSE #1
```

Figure 5

Two new commands have been introduced here—GET and PUT. Unlike their graphics counterparts (which have no relation to these file versions), GET and PUT are used to read and write disk records.

TIP

One of the nice features of QuickBASIC has always been the REDIM command. If the array hasn't already been dimensioned it will be created at that point in the program. And if it already exists, it will be erased and then dimensioned again without causing an error. Unfortunately, Turbo Basic doesn't have REDIM. The best approach we have found is to DIM every array that may need to be REDIM'ed later to 1 element at the beginning of the program. Then, always use ERASE immediately before any subsequent DIM statements:

```
DIM DYNAMIC Array(1)
.
.
.
ERASE Array: DIM Array(size)
```

Besides indicating which file number is to be read or written, you also specify which record number to operate on. In truth, the record number is an optional parameter, and if it is left out, BASIC will default to the next one in sequence. Personally, I usually include an explicit record number, just to eliminate any possibility of a mix-up. Two other new commands being used here are CVI and CVS, which complement MKI$ and MKS$ respectively. Where MKI$ obtains the two bytes that comprise an integer value and creates a string from them, CVI does exactly the opposite. That is, it takes a two-character string and creates an integer value from the characters.

CVD and CVL also convert strings to numbers, with the first intended for double precision values, and the second for long integers. Like the MKI$ example shown earlier, the action of CVI can also be imitated by normal BASIC commands. Of course, I'm not recommending that you program this way, but in the interest of completeness, here's what CVI really does:

```
X% = CVI(Value$)
X% = ASC(Value$) + 256 * ASC(RIGHT$(Value$, 1))
```

RANDOM FILE TECHNIQUES

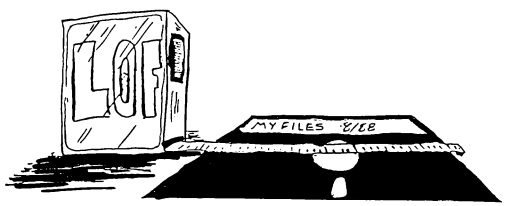
Now that we know the essential operations that are needed to manipulate random access files, lets look at a few real life situations. After a file has been created and data placed into it, one of the first things you'll surely want to do is be able to report on that data. For example, you may need to identify all accounts that have had a balance due for more than 30 days. Or perhaps you want the ability to delete records from the file, or provide other reporting options. We're not going to pursue a lengthy discussion of indexing or sorting techniques here, however a few practical examples come to mind.

One of the first things you'll need to know is how many records are in the file. This is easy to determine by dividing the file size by the record length:

```
OPEN "MyStuff.Dat" FOR RANDOM AS #1 LEN = 87
NumberOfRecords = LOF(1) / 87
```

Notice that LOF is a handy way to obtain the length of any file, including .COM or .EXE programs. But be careful to close the file as soon as you get its length, to avoid any possibility of altering it:

```
OPEN "AnyFile.Ext" FOR RANDOM AS #1 LEN = 1
Size = LOF(1)
CLOSE
```



Most of the data you'll be storing in a random access file will consist of either strings or numeric values, however the best way to store certain information is not always obvious. For example, dates can be represented in a variety of ways. At the minimum, you should omit any separating hyphens or slashes, and only show them on the screen that way. That is, 01/15/88 could be kept on disk in a six-byte field as 011588. But this method does not allow a direct comparison; it is not obvious that 011588 is later than 123187 regardless of whether you use a string or numeric comparison.

A much better approach would be to swap the digit pairs around so that the year comes first, followed by the month and day. Since one of the main objectives of a database report is to process the information as quickly as possible, using this technique will provide a dramatic improvement.

```
OPEN "Accounts.Dat" FOR RANDOM AS #1 LEN = 125
NumRecs = LOF(1) / 125

FIELD #1, 32 AS AccountName$,  _
          32 AS Address$,      _
          25 AS City$,         _
           2 AS State$,        _
           5 AS Zip$,          _
          14 AS Phone$,        _
           6 AS DateDue$,      _
           1 AS PaidYN$,       _
           4 AS LastPayment$,  _
           4 AS BalDue$

Today$ = RIGHT$(DATE$,2) + LEFT$(DATE$,2) + MID$(DATE$,4,2)

FOR X = 1 TO NumRecs
   GET #1, X
   IF DateDue$ <= Today$ AND PaidYN$ <> "Y" THEN
      LPRINT AccountName$, Phone$, CVS(BalDue$)
   END IF
NEXT
CLOSE #1
```

Figure 6

A database report that examines every record in the file.

The example in Figure 6 opens a hypothetical disk data file, and then lists the name, phone number, and amount for all accounts that are due but not yet paid. This example assumes that the field PaidYN$ will contain either a "Y" if the account has already been paid, or an "N" or a blank if it has not. We'll also assume that the dates in the file were swapped around into a YYMMDD format when each record was written. If the date the account was due is today or earlier and it has not already been settled, then the name and other information will be printed. Since you'll undoubtedly be coding these date fields many times, this is a natural application for BASIC's multi-line user defined functions. In fact, an even better approach is to pack all dates into only *three* bytes to save disk space, using CHR$(Year - 1900) + CHR$(Month) + CHR$(Day).

Of course, the QuickPak date conversion routines can provide an even greater savings by packing a date into only two bytes.

DELETING RECORDS

The last item we'll consider is a method for deleting records from a database. Of course, there's no reasonable way to physically remove a record from a disk file, so our only recourse is to mark it in some way. Many commercial database programs reserve an extra byte in each record for exactly this purpose, however with some clever programming we can eliminate that wasted byte.

DOS-O-MANIA

Boot is not kicking your computer.
Byte is not what your dog does to the mailman.
Root is not the book Alex Haley wrote.
Reboot is not kicking your computer again.
Park is not what you do at Inspiraton Point, and
Drive is not how you got there.
Function keys do not open any doors.

Since text fields such as a name or address don't need to accommodate the PC's extended graphics characters, the simplest approach is to convert one of the letters in a name to its corresponding graphic symbol. This is accomplished by either adding 128 to the character's ASCII value, or by using the BASIC OR function to do the same thing. The example below retrieves the record to be deleted from the file, adds 128 to the ASCII value of the first character in the last name field, and then writes the record back to disk.

```
GET #1, RecordNumber
LSET LName$ = CHR$(ASC(LName$) + 128) + MID$(LName$, 2)
PUT #1, RecordNumber
```

Then, when you are reporting on the file and need to tell if a record was deleted and should not be included, all you have to do is check the ASCII value of the field:

```
GET #1, RecordNumber
IF ASC(LName$) => 128 THEN . .    'record is deleted
```

SUMMING UP

We have looked at a variety of techniques for reading and writing random access disk files, as well as several tips and techniques you can apply in your own programs. While these examples are far from the final word on the subject, I hope they will encourage you to experiment on your own, and further explore one of BASIC's most powerful capabilities.

TIP

Most printers have a hidden feature that lets you underline or emphasize text by issuing a carriage return, but *without* the usual line feed. If you print a CHR$(13), BASIC stupidly steps in and sends an unwanted CHR$(10) automatically. One way around that is to instead use a CHR$(141). Many printers ignore the high bit of each character and simply interpret a 141 as a 13.

```
LPRINT "This is underlined"; CHR$(141); STRING$(18, 95)
```

But this doesn't work with a LaserJet or other printers that accept all 256 characters. The only *sure* way is to use the QuickPak BLPrint routine that bypasses BASIC altogether:

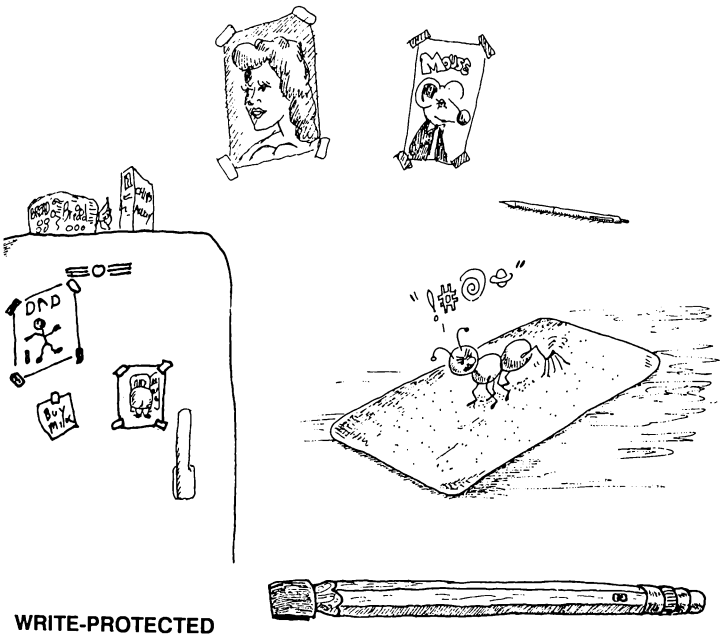
```
CALL BLPrint(1, "This is underlined" + CHR$(13) + _
     STRING$(18, 95), 0)
```

Just a few of our many satisfied customers . . .

American Suzuki
Anheiser Busch
Ashton Tate
Arco
Bendix
Boeing
Campbell Soup
Celanese
Cadbury USA
Chevron
Ciba-Geigy
Clorox
Coca Cola
Compaq
Delco
Dow Chemical
Dupont
Eastman Kodak
EG&G Ocean Product
Eli Lilly
Exxon
Ford Motor
Gillette
General Electric
Goodyear Tire and Rubber
GTE
Hanes
Hewlett-Packard
Hughes Aircraft
Ingersol Rand
International Paper Co.
Internal Revenue Service
Martin Marietta
Metropolitan Life
Monsanto
NASA
National CD Network
National Weather Service
Nynex
Olin Corp
Pacific Bell
Parke-Davis
Perdue Farms
Prentice Hall
Proctor & Gamble
Radio Shack
Rand McNally
Reynolds Metals
Shell Development
Stouffer Restaurants
Techna Vision
Tektronics
3M Corporation
Texaco
Touche-Ross
Uniroyal Goodrich
UPS
US Army
US Dept. of Agriculture
US Navy
Vicks
Wang Labs
Warner-Lambert
Westinghouse
Xerox Systems
Yamaha international

WHAT'S WRONG WITH THIS CODE?

```
GET Lost,, Baby
POKE Your, Eye
CALL Home
CHAIN "Fence"
OPEN "Sesame"
STATIC Electricity
VAL(Girl$)
EXP OSE!
```



WRITE-PROTECTED

What good are write-protect tabs anyway? Well, for starters you can use them to seal an opened can of Pepsi. Also, nerds will find them indispensable for repairing their eye glasses. You could even scatter them around your desk sticky side up—they make great ant traps!

Challenges for the Future

by Gregory E. Lobdell
Microsoft Corporation

BASIC: THE ROOTS OF PERSONAL COMPUTER LANGUAGES

BASIC, along with the rest of the personal computer languages, has evolved dramatically since the MITS ALTAIR was introduced in 1975. Languages in general are driven by several factors, including market pressure and standards committees. Formal languages must be defined, refined, and enhanced to meet the needs of those using them. In addition to formal languages, the environments built around the languages has evolved as well. Integrated editors, debuggers, and other tools have become commonplace.

As hardware prices have dropped well below $1,000 for entry level systems, and powerful easy-to-use computer languages are available for under $100, more and more enthusiasts have entered the ranks of programmers. Programmers have come to expect ease-of-use, integration, and a friendly user interface, while not sacrificing a powerful language implementation.

With the rise of the 80286 and 80386 microprocessors, the availability of powerful operating systems such as Microsoft OS/2, and low-priced entry-level systems, languages are ready for another in a series of leaps forward—leaps that will make the language marketplace one of the most fascinating, competitive, and high growth areas of the personal computer industry.

TIP
Wanna convert your compiled BASIC .EXE programs to .COM files? Easy, just rename them. (No fooling, it really works!)

```
REN PROGRAM.EXE PROGRAM.COM
```

In the last year, we at Microsoft have talked about the major areas where we expect the next leaps to occur. These are:

1) Delivery of traditional languages which will allow previous non-programmers to master the power of BASIC (and other languages).

2) Integration of BASIC and traditional applications.

3) Evolution of computer languages, in general, toward an ideal language.

THE EVOLUTION OF BASIC–MEETING THE NEEDS OF THE MARKET

When looking at the languages marketplace there are three very interesting segments: the professionals, the enthusiasts, and the non-programmers.

The professionals are perhaps the easiest segment to quantify, they are those who earn their primary income writing software; either for retail sale or for use within their organization. Understanding what is important to this segment is relatively easy–they are typically a very vocal crowd, willing to let a vendor know how he can improve his product. Usually, power and speed are their primary concerns. That is, speed of development and execution, and the power of the underlying language.

The enthusiasts are a more subtle segment. Coming from a wide background, they use a language as a tool to perform some specific mission or task, ranging from utilities to small business applications. Productivity is key to these individuals.

Finally, the "non-programmers" are perhaps the most overlooked segment in the market today. This segment is comprised of the millions of PC users who employ the PC as a tool to better perform a series of tasks, whether at home or on the job. They have never had the motivation to learn a traditional programming language like BASIC, yet they

often do pseudo programming with tools like the DOS batch language, dBASE, Excel or Lotus 1-2-3 macros, and possibly a little GW-BASIC or BASICA. While these people think of themselves as non-programmers, they have typically performed a number of programming-like tasks.

When looking at these segments, it is our belief that it is impossible to provide a single product which meets the needs of all segments. In the future you will see greater differentiation in the Microsoft language family between the products targetted at the professional programmers, and those targetted at the enthusiasts and novice programmers. Our challenge is to develop products which meet the needs of the various segments.

The design philosophy for future entry-level languages will be ease-of-mastery. That is, providing a tool which helps the new programmer become productive as quickly as possible. In addition, the technology should not hinder the growth of the beginning programmer or the more advanced programmer.

The underlying technology will be enhanced to improve the learnability of the products. Inovative technology will be added that will put more information at your fingertips, both for learning and for reference purposes. The environment will be enhanced to become a dynamic system that grows with you as your programming skills grow.

For the professional products, you will see two primary areas of change. First, the development environment will evolve into a common development platform, offering the integration found today in the "Quick" languages, without the limitations. Although OS/2 will provide the long term systems software platform, an integrated development environment will be provided for the MS-DOS developer. This platform will integrate all aspects of the development process, from editing to debugging. A natural progression of our common-tools strategy (CodeView and the Microsoft Editor), new tools will be added and considerions such as work-group development on LANs will be key components.

In addition to the development environment, the core technology of the products will continue to be a key area of enhancement. Code generation and optimization technology will continue to be evolve. There will also be some logical next steps—pointers in BASIC for example. Object-oriented extensions will enter both the BASIC and the C languages, and specific support for graphical user interfaces (such as Windows and Presentation Manager) will be added. These all center around our goal of providing the professional developer the tools necessary to develop outstanding applications for MS-DOS, Windows, and OS/2 with Presentation Manager.

INTEGRATION OF COMPUTER LANGUAGES TO APPLICATIONS

There are really two categories of computer languages. The first is the traditional languages such as BASIC, FORTRAN, Pascal, C, COBOL, and many others. These languages are designed to give the application developer the ability to tap into the PC's resources, both the operating system and the hardware itself. Each of these languages have particular traits which make it more (or less) suitable for solving particular programming problems.

An engineer uses FORTRAN for solving difficult mathematical equations, while the business programmer uses COBOL with its powerful file handling. The systems level programmer uses C for reaching into the depths of the system and tapping into the lowest level of system resources.

Although "cult" followings have emerged, most sophisticated programmers will agree that each language is useful in specific applications areas.

MACROS ARE A LANGUAGE TOO

The second category is one that is not typically thought of as computer languages. These are the languages embedded in the various application products. The macros in Microsoft Excel for the PC and the command-language of Lotus 1-2-3 are prime examples of programming languages embedded within applications. Actually, if you step back and look at any computer application, virtually every application is a language processing system. They take commands (the language itself) and perform operations based on their interpretation of the command.

Traditional computer languages are designed to tap into the PC's system resources. In a similar fashion, the embedded languages within applications are designed to tap into the resources of the application they are embedded within.

However, these embedded languages have severe limitations, particularly when you compare them to their counterparts in traditional languages. Initially, these macro languages were designed to merely playback a series of keystrokes. As they have evolved, they have added some simple control structures (for example, IF-THEN-ELSE) but they are still not nearly as powerful as traditional computer languages. Macros created in one application cannot be used by another. In addition, there is no accepted method for macros from one application to talk with macros in another application.

One final limitation is that there is no way for the embedded macro languages to tap into the system resources of the operating system, or, in most cases, for traditional languages to tap into the resources within applications. This limitation has led to a huge amount of duplication of effort. Hardly acceptable under MS-DOS, this will be intolerable under OS/2 and the Presentation Manager. Because of the multitasking nature of OS/2, users cannot be expected to remember three completely different sets of macro commands when running three applications at once. In addition, why should resources within applications be accessible only from within that application?

A COMMON MACRO LANGUAGE:
A NEW WAY TO PROGRAM

The barrier between what resources are inside an application and what's outside are largely artificial, imposed upon us by the way the application is packaged. Ideally, one would like to see an environment in which all applications, once installed into the system, become part of a common pool of resources. With this architecture, the barrier described above is demolished.

With this barrier gone, the way is paved for a common macro language. This new architecture will allow programmers to use a common language. This language, probably built upon a well-known syntax like BASIC, will allow programmers to access the powerful resources found within applications. Coupled with an efficient and robust common protocol for communication between applications, a common macro language could open up possibilities never before imagined.

Along with tapping into the resources within applications, a common macro language could be built into the operating system, replacing the old batch language and simple BASIC programs often used for housekeeping tasks. Benefiting from a common language, users could tap into the system or into the applications, something no current language can do.

It should be noted that a common macro language is not a replacement for traditional languages or applications. Applications, designed to perform a specific set of tasks, will continue to be the cornerstone of the PC revolution. A common macro language merely provides a mechanism to tie these applications together more closely.

THE EVOLUTION OF AN IDEAL LANGUAGE

The second area is the evolution of today's computer languages toward an "ideal" language. By "ideal" we do not mean to imply that there will someday be a single language that meets the needs of all programmers. Rather, there are certain characteristics of various languages, that if incorporated, could move us in the direction of an "ideal" language.

For illustration, a simple definition of an "ideal" language is a language that makes programming extremely easy for the beginning user, while providing the power and flexibility required for serious applications programming.

Using this simple definition as our basis, it makes sense that this "ideal" language will evolve from an existing, very well-known, and easy-to-use language such as BASIC. BASIC is an obvious choice because it is known by virtually everyone who owns an MS-DOS computer. In addition, it is a great example of a language in evolution. The past several years have seen powerful control structures and modular programming features added to the language. With the release of Microsoft QuickBASIC 4.0, true user defined types and record structures finally bring "structure" to the data as well.

This "ideal" language should feature the "interactiveness" of an interpreter, with the speed of a compiler. The current generation of languages on PCs have dramatically increased the compile speed, but have not provided the ease-of-use associated with an interpreter. Microsoft QuickBASIC 4.0 represents the first major step in this direction. The threaded p-code interpreter in QuickBASIC 4 provides all of the "interactiveness" found in earlier BASIC interpreters, as well as some capabilities beyond the older interpreters, while providing performance close to a true compiler.

One final point about an "ideal" language. An "ideal" language will evolve in a direction that both makes it more 4GL-like as well as more object oriented. We are already seeing this in the C world with C++ becoming an important part of C compilers. Object oriented programming has not caught on as fast as some thought it would because there has not been a great requirement for it. However, in the Presentation Manager environment, object-oriented programming not only is a perfect fit, but makes some tasks easier and more intuitive. When you think about a graphical user interface, it is a collection of objects with which the user interacts. The control of these objects is an ideal application of object oriented programming.

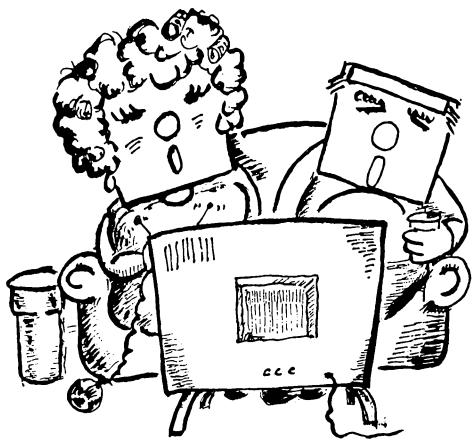
Although QuickBASIC 4.0 has many of the features of an "ideal" language, these capabilities can be designed into any language. In fact, many of them will. Each language will continue to have specific groups of followers, and each will be enhanced to move in the direction of this "ideal" language.

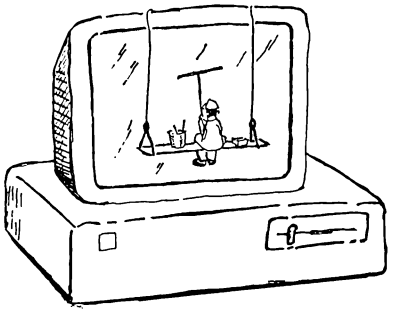
DOS-O-MANIA

CD is not what you spent megabucks on to replace your tape deck.
MD is not who asked you to "fill the little bottle please".
Batch processing is not done with cookie recipes.
User friendly isn't what you think it might be.
Commands are not what you yell at your dog.
You can't get a directory by dialing 411.
Chkdsk is not an Eastern European country.

CONCLUSION

We have looked at the challenges of providing products which meet the needs of several categories of potential users, the integration of languages and applications, and the evolution of the languages themselves toward an "ideal" language. The low-entry price of hardware and the power provided in the new hardware and operating systems is opening up some exciting doors for the application designer and solution-oriented PC user. It is our challenge to step through this door and provide outstanding development tools and environments for the creation of the next generation of applications; as well as tools that bring new meaning to "ease-of-learning" to the millions of non-programmers who can solve more problems, and be more productive with a language like QuickBASIC. Fortunately, these advances are no longer merely a vision, but will soon be a reality.





MORE USELESS WAYS TO CLEAR THE SCREEN

```
LOCATE 1, 1: PRINT SPACE$(2000);

PRINT CHR$(12);   <believe it or not 'C' programmers use this.

LOCATE 25: PRINT STRING$(24, 13)
```

TIP

There are lots of handy shortcuts that even the experienced programmers don't know. All those shown below will reduce the number of keystrokes needed at the DOS command line.

```
DIR .BAS      (no leading asterisk necessary)
DEL A:\       (A:\ is the same as A:\*.*)
COPY A:\ C:   (copy all files in root directory on A: to C:)
DIR ..        (get a DIR of the directory above)
CD..          (change to the directory above)
CD..\OVER     (move to a directory one level over)
CD..\..       (move to a directory two levels up)
DIR *.        (show only files with no extension)
DIR A*        (show all files that begin with "A")
```

While you're at it, don't forget that the F3 key will retype the last command you entered.

BASIC Tools from Crescent Software

WHY TOOLS?

Programming tools and utilities have traditionally been an important addition to many programming languages. Although languages such as C *require* external products to do anything useful, BASIC has always enjoyed a rich endowment of built-in commands.

So why should a BASIC programmer consider purchasing add-on products? There are several good reasons. One obvious advantage to purchasing tools is the enormous savings in time and effort to achieve a completed program. Though many BASIC programmers may be technically able to develop a sophisticated pulldown menu with full mouse support, many weeks of work are required. It simply makes more sense for the professional programmer to purchase the necessary routines.

Another reason is that many of the tools we offer are written in assembly language. Thus, you can achieve results in your programs that would not be possible using BASIC alone. For example, QuickPak Professional comes with routines for searching and sorting all of the QuickBASIC 4 data types. These are typically hundreds of times faster than an equivalent program written in BASIC. Moreover, assembler routines greatly simplify access to DOS and BIOS services. A quick look at the QuickBASIC and Turbo Basic manuals will confirm how difficult this would be without expert assistance.

Yet another important advantage is code size. Most of the assembler routines offered by Crescent Software add less than fifty bytes to your programs. Any DOS service performed with BASIC's limited CALL INTERRUPT would require hundreds of bytes and of course be much slower.

Finally, there is the educational value—all of our products include fully commented source code. We provide the source not only so you can modify the programs to suit your needs, but also because we want you to see how they operate. Most programmers will readily acknowledge that the best way to learn more about programming is to study other people's work. All programs that we produce are expertly written, and include extensive comments.

ADDING OUR ROUTINES TO YOUR PROGRAMS

Combining our routines and functions with programs of your own is easy. Products such as QuickPak and QuickPak Professional that include assembler routines come with two library files, each containing all of the routines. One of these is a Quick Library for use with QuickBASIC in the editing environment, and the other is a linking library that is used to create a final .EXE program.

To load a Quick Library into the QuickBASIC editor simply start it with the /L option and the name of the library:

```
QB /L QuickPak
```

Once this is done, all of the routines are available to be called by your program. For example, to set the current default drive to Drive A you would add this statement to your program:

```
CALL SetDrive("A")
```

or

```
Drive$ = "a"
CALL SetDrive(Drive$)
```

BASIC routines are just as easy to add. For QuickBASIC 2 and 3 simply add this line to your program:

```
'$INCLUDE: 'filename'
```

QuickBASIC 4 lets you have multiple source files in memory at once, so all you have to do is load the ones you want. It's as easy as that!

Each of our products is described fully on the pages that follow. If you have any questions or need further information, please call us. We are constantly adding new capabilities and features to all of our products. Also, because of the sheer number of routines included with many of these packages, it is impossible to list every one. If you need a program or feature that is not mentioned here, there's still a good chance that we have it. Again, we encourage you to call for advice on any programming problems you need help with.

All of our products are available for QuickBASIC, and most of them for Turbo Basic as well. It is *essential* that you specify not only which brand of BASIC you are using, but also the version number. In many cases, we have separate versions that have been optimized for each particular compiler.

HOW'S THAT?

With many BASIC keywords and characters, it's not always obvious how they should be pronounced. For example, we've heard people say "X-Dollar" for X$, when "X-String" is so much easier to say. Another gem is VARPTR. The folks at Microsoft say it "Var-Put-er", but we greatly prefer "Var-Pointer" because, after all, it's a pointer function.

Along the same lines, a lot of folks say "Lib" as in "ad-lib" for the LIB.EXE program or when referring to a file such as QUICKPAK.LIB. Because LIB is short for LIBRARY, we prefer to pronounce it with a long "i". Microsoft agrees, as do the authors of the db/LIB database extensions for QuickBASIC.

But how would you pronounce CHR$(13)? Simple—"Character String Thirteen" (or "Character Thirteen"). To our ears, that's better than "Char Dollar" any day. Likewise, INSTR is "In-String", and LEFT$ should be "Left-String".

QuickPak ($79)

QuickPak is a collection of more than 65 "toolbox" routines for both beginning and advanced BASIC programmers. Approximately half are written in assembly language to perform functions that BASIC either cannot do directly, or is impossibly slow at. The BASIC routines provide a variety of services that would be difficult or tedious for most programmers to write themselves.

Included are programs for windowing, access to DOS and BIOS services, searching and sorting string arrays, creating pull-down and Lotus-style menus, accepting data input, and much more.

QuickPak includes sample programs and full source code for every routine, tutorial information, and The Assembly Tutor—an introductory guide to learning assembly language from a BASIC perspective.

All of the QuickPak routines are designed for easy use. The number of parameters required is kept to the absolute minimum, and complete instructions are included for each program. Because there are so many different programs included with QuickPak, there is room to list only some of them here. These programs fall into five general categories—DOS, String, Video, Input, and Menu.

The DOS services permit operations not possible using BASIC alone. For example, you can obtain a list of file names from disk, set and change the default drive, determine the total and free space remaining, and read and write sectors directly.

Many useful string routines are provided such as Sort, Find, Encrypt, and PUsing. Sort will alphabetize all or part of a string array *very quickly.* Find lets you search an entire array without regard to capitalization and supports wild cards. Encrypt lets you protect sensitive data by encoding a string with a password you provide. PUsing mimics BASIC's PRINT USING, except it returns the formatted value in a string.

Video routines include windowing, painting, and two very fast display routines. One is intended for printing a single string, and the other is meant for string arrays. The array routine is particularly useful and greatly simplifies the creation of a "browse" facility, where the entire array may be scrolled up or down, left or right.

Several input routine are provided to allow entering and editing strings, numbers, and dates. Besides letting you control the length of a string to be entered, you may specify that all letters be converted to upper case, or accept only numbers.

Five separate menu routines are included with QuickPak. The pulldown menu is similar to those in the QuickBASIC and Turbo Basic editors—simply define a list of choices for each menu, and the program does the rest. Two menus use the familiar scroll-bar method for selecting items, and two others use the Lotus 1-2-3 approach for displaying choices and prompts on a single line.

QuickPak Professional ($149)

QuickPak Professional is the most comprehensive set of tools and utilities for BASIC programmers ever developed. It includes all of the routines in QuickPak, plus many, many more. More than three hundred separate routines are included, not counting dozens of demonstration programs.

All of the QuickBASIC 4 data types are supported, including fixed length and TYPE variables. Extensive documentation is provided with tutorials on files, arrays, subprograms, sorting and more. Of course, complete BASIC and assembler source code is provided.

Because of the sheer number of programs included with QuickPak Professional, it is impossible to list them all here. However, many of the highlights are described below.

A full set of file and printer routines are provided to eliminate the need for ON ERROR. Many programmers prefer to avoid BASIC's error handling, because the resulting .EXE program will be smaller and run faster.

Other low level routines provide string and file encryption, file and array searching, date and time calculations, access to mouse services, and extensive string manipulation. A sophisticated screen dump program is included that operates in *any* graphics mode and works with any printer that accepts the IBM/Epson or HP LaserJet control codes.

All of the assembler routines that process strings and string arrays are provided in an alternate version that ignores capitalization. All of the video routines operate on any screen page, and support the 43 and 50 line EGA/VGA modes automatically.

QuickPak Professional also includes many complete applications that may be added to your programs. Dozens of callable BASIC modules are provided including a complete spreadsheet, and a full-screen editor with word wrap, block operations, and mouse support. Also included are a pop-up calculator, calendar, ASCII chart, and a browse program that handles files of nearly any size.

QuickPak Professional contains a comprehensive set of scientific and financial functions, including *all* those offered in commercial spreadsheet programs. Other major utilities are a complete pull-down menu system with mouse support, vertical menus that accommodate any number of choices, and a recursive TYPE array sort that can handle any number of keys. The editor and menu programs can be used either normally, or in a unique "multi-tasking" mode that lets you display several menus at once, and poll them in sequence.

Many programs are provided for sophisticated window handling. The window manager accommodates ten levels of window nesting, and may be easily expanded to handle any number of screens. A special assembler routine also lets you close just a portion of a larger window.

A string manager is included to let you quickly copy an entire string array out to far memory and back again, and a unique pause routine allows you to create delays with microsecond resolution. A full featured expression evaluator is also included that accepts complex expressions with any level of parentheses, and returns the resulting value.

QBase ($149) and Quick Screen ($79)

QBase is a revolutionary concept in BASIC screen and database software, not only because it's so easy to use, but also because it does so much. Where most screen builders merely create a screen image to be displayed by your program, QBase lets you define data entry fields, create custom help screens, and even manage an entire relational database.

This means you can keep a list of customers in one database, and a series of transactions in another. Then, whenever a sale is entered, the program will retrieve the name and address from the customer file automatically. Relations may be established between any files in the database, based on a match between like fields.

QBase can do all this because it's really two separate programs—a screen builder and a run-time application. If you prefer to write your own routines to handle file I/O and indexing, then simply purchase the screen builder and data-entry module.

The QBase screen designer is available separately, and includes all of the routines to display screens, and enter and edit input fields. Unlike other screen builders, Quick Screen (and QBase) do *not* require any memory resident modules to display screens. Rather, we provide BASIC and assembler routines that are added to your program directly.

WHAT IT ISN'T

QBase is *not* a code generator. The main problem with code generators is that they can never do exactly what you need. Worse, you end up with a separate large .EXE program for each screen. Once a code generator has created a program, you'll undoubtedly have to make changes. But once you've modified the code, it's difficult to go back and alter the original screen, because you'll have to add the changes again. Further, most BASIC code generators create programs that are poorly organized and use line numbers.

THE QBASE APPROACH

QBase avoids that mess by saving screens and fields as *form definitions.* These definitions are logically arranged and kept in disk files, which are simple to access from any program. Multiple screens are contained in a single library file that is loaded once at the start of a data entry session. Then as each screen is needed, it can be displayed very quickly. Other libraries may be loaded as needed, allowing an unlimited number of screens in a single program.

But the real advantage of QBase is the supplied database program. Since a single program is used for all of the databases, any modifications you make will be reflected in all of the applications you create. For example, if you wanted to re-assign the function keys, you only need to change the program once. This means that you can even modify an existing database, without having to go back to the original screens. And if you ever need to add, delete, or change fields, a "rebuild" facility is included that will convert an existing data file to the new format automatically.

HOW IT WORKS

The screen builder features pulldown menus, multiple help screens, and function keys that are logically organized. Separate menus are used to load and save screens, select colors and line types, draw boxes, and cut or copy blocks. Shortcut keys are also provided for most of the operations if you prefer.

Defining fields is as simple as pushing a key, and answering a few questions. You may specify string, numeric string, integer, single or double precision field variables, as well as date, currency, logical, and multiple choice.

The multiple choice fields are especially useful, because they eliminate extra typing and spelling errors. For example, if you're creating an accounting system that must keep track of expenses by category, most databases would require you to define the category field as text.

Now suppose the data entry clerk types "Payroll" one day, but goofs the next time and enters "payrll". Good luck trying to balance your books!

QBase instead lets you establish a list of acceptable choices, which are displayed in a pop-up menu when the cursor is on that field. Only the choices shown will be accepted, and even better, the choice number is stored in the file as a single integer word. This affords a tremendous reduction in disk space compared with storing the full text of each choice.

The QBase database is truly a full-featured program. It allows stepping forward and backward through the file, searching based on multiple criteria, and adding and updating records. More than fifteen screens can be included in a single database, and each screen may contain up to one hundred fields and 32,766 records. A standard help screen is always available, though it may be edited just like any other QBase screen. You can also create custom help windows for each entry screen.

WHAT'S INCLUDED

QBase comes with a screen builder, database, and file rebuild programs, a help window demonstration program, sample screens and applications, and a slide show program which displays a series of screens automatically. Quick Screen is identical, except the database program is not included.

QBase includes fully commented BASIC source code to show how it works, and where it may be customized. The assembler source requires QuickPak Professional, which is available separately.

As a special bonus, we've added The Hardware Tutor—a clear and practical discussion of electronic concepts and circuits, written from a programmer's perspective.

QBASE MAXIMUM LIMITS

Records per file	32,766
Fields per form	100
Indexes per form	5
Files open at once	5
Relational fields	no limit

QBASE PERFORMANCE TEST

The following tests were performed on a stock IBM AT with a 30MB hard disk, using a "real life" data file comprised of 21,000 records. All timings indicate how many seconds it took to locate the last record in the file, based on the field type. For smaller files, the times will be proportionately less.

The actual search times will vary for string fields, depending on the number of similar records. For example, locating the last "Jones" will generally take longer then the last "Szabo", because more records have to be examined. Notice that when date or integer fields are used (part numbers, account numbers, and so forth), the response time is nearly instantaneous.

Indexed Searches

Using an integer field	0.03
Using a date field	0.03
Using a string field	8.00

Non-Indexed Searches

Using any field type	77.00

QBase Report ($79)

QBase Report is meant for use with QBase data files, and offers three kinds of reporting capabilities. The conventional report type lets you summarize the contents of a file sorted on two key fields, with the output directed to the screen, a printer, or disk file. The Modify report type allows updating selected records in a file, for example flagging accounts that are more than thirty days overdue. The Add reporting option can be used to copy all or selected records from one data file to another. If needed, field type conversion will be performed automatically during record adding or modification.

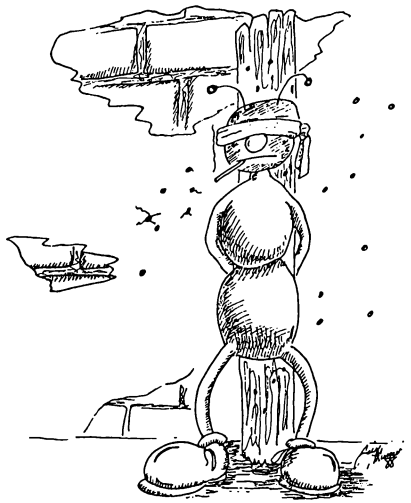
A simple command language is used to indicate the primary file for the report, as well as any related files to be included. An Assist mode is provided to guide you through the report's design. Reporting options include headers, footers, margins, totals, and four-function math. Also supported are string and numeric constants, and current date, time, and page number fields.

QBase Report lets you chain multiple reports to perform several functions in sequence without any user intervention. A unique Browse mode features a clever multitasking scheme, whereby selected and sorted data may be viewed even while it's being processed!

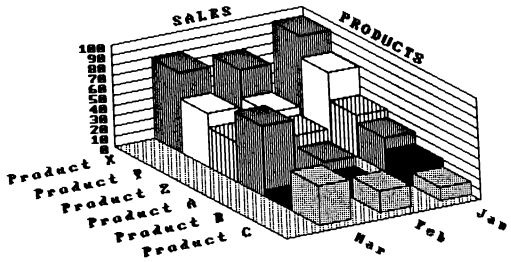
An on-screen report layout feature lets you quickly indicate how the report is to appear, and where each field will be positioned. You may also specify that the report is to pause before running, allowing the user to specify either the screen, printer, or disk file.

QBase Report includes three additional bonuses—a complete time billing application designed for computer professionals, a stand alone file sort utility, and an import facility for QBase data files.

All BASIC source code is provided. The assembler source requires QuickPak Professional, which is available separately.



Bug Fix # 132



GraphPak ($79)

GraphPak is a collection of subprograms for displaying attractive 3-dimensional bar, pie, and line graphs from within a BASIC program. All of the popular display adapters are accommodated automatically, based on the type of monitor that is detected when the program runs. GraphPak can also be used to display text, titles, and legends in any size, color, or style.

GraphPak is comprised of both high-level routines and a number of low-level primitives. The high-level routines are responsible for accepting the numeric data and text to be displayed, which are then used to calculate the appropriate scaling, angles, and colors automatically.

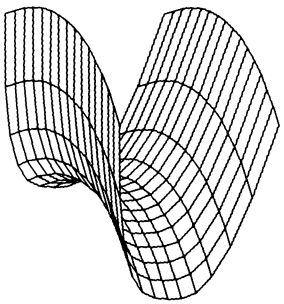
Individual low-level routines are also provided to create textured backgrounds, draw graphs, position and print text, and perform scrolling and windowing in the popular graphics modes. By supplying both high-level and low-level functions, the BASIC programmer is given with the ability to display complex graphs very quickly, as well as complete control when necessary.

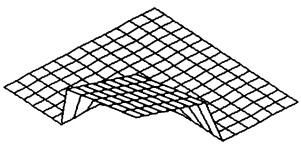
GraphPak also includes a font and tile editor for creating custom character sets and attractive backgrounds, and a subprogram to send high-resolution EGA graphics to an HP LaserJet printer. All BASIC and assembler source code is included.

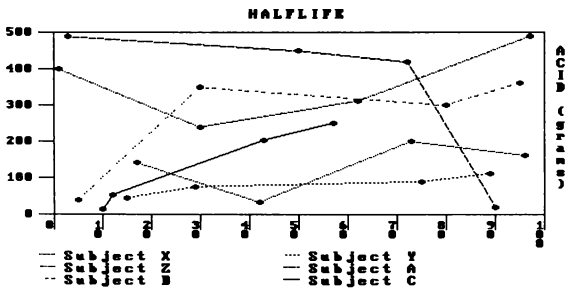
GraphPak Professional ($149)

GraphPak Professional improves on the original GraphPak by adding many important features including business and scientific graphs, surface plots, and the ability to display multiple fonts simultaneously. Also added are text-based bar charts which are displayed very quickly on *any* type of monitor.

As a special bonus we have included a screen print routine that works with *any* screen mode and any graphics printer that supports either the LaserJet or the Epson/IBM control sequences. All BASIC and assembler source code is included.







LaserPak ($79)

LaserPak is a complete set of BASIC subroutines for controlling an HP LaserJet or compatible laser printer. Modeled after the BASIC graphics commands, LaserPak allows BASIC programs to quickly and easily draw lines, boxes, circles, and fill and shading patterns. Other capabilities include graph scales and grids, text labeling, and complete control over all of the LaserJet control sequences.

LaserPak also includes a sophisticated symbol editor that can be used to design and manage logos, clip-art, and custom fonts. A complete demonstration program is provided that shows all of the LaserPak features in context. All BASIC and assembler source code is included.

LaserPak Professional ($149)

LaserPak Professional enhances the original LaserPak by providing support for dot matrix printers, as well as a number of other new and important features.

Highlights include a unique "buffered printing" output capability. Normally, the amount of data that can be sent to a laser printer depends on how complex the images are, and how much memory is installed in the printer. The buffered printing feature instead calculates the complete image ahead of time, and sends it in as few bytes as possible.

The pattern and symbol editor has also been enhanced to provide full support for a mouse, and an improved resolution of 150 by 150 dots per symbol. New subprograms include open, filled and partial ellipses, pie portions, and three-dimensional bars. Bars and ellipses may be freely superimposed, and LaserPak Professional will print the images in the correct order. Thus, hidden lines are always handled correctly.

Output for all of the subprograms may optionally be sent to a file for printing at a later time.

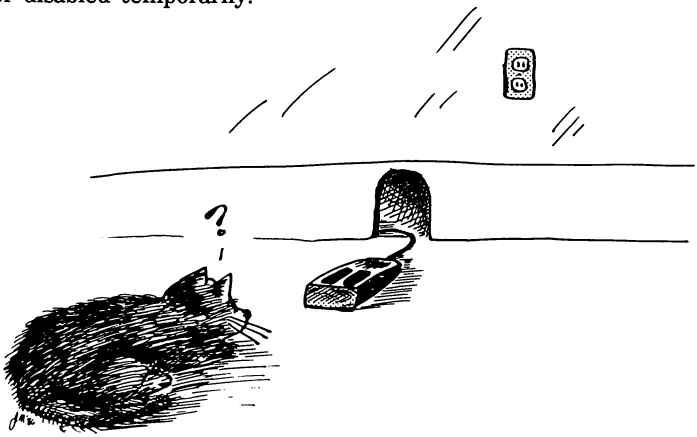
QuickHelp ($69)

QuickHelp is a complete help message system that lets you add instant pop-up help to any application. Unlike other help products, QuickHELP requires only 22K of DOS memory. This is less than *one third* the amount of memory needed for the "other" popular on-line help package.

QuickHelp features indexed file access for fast retrieval, a user-definable hot key, and customizable colors. It accommodates the 25, 43, or 50 line display modes automatically, and comes with a compiler for creating an unlimited number of help screens.

Rather than require you to navigate through menus to find the information, QuickHelp provides help for the word at the current cursor location. This is ideal for use with our QuickPak Professional product, which includes more than three hundred separate routines.

The QuickHelp information is kept in a disk file in the current directory. This lets you have multiple help files available, no matter which application you are using at the time. QuickHelp may be unloaded from memory if needed, or disabled temporarily.



Creating your own help messages is extremely easy, as the example below illustrates.

```
@Keyword-1
This is the text that will appear when QuickHelp is activated
and the cursor is on the word "Keyword-1".  The QuickHelp
text window will be sized automatically based on the length
of the longest line, and the number of lines in this block.
@Keyword-2
This is the help that will appear
for the word "Keyword-2".  As you
can see, the width of the window
is controlled very easily.
```

Once the help text has been created (using any ASCII text editor), simply run the supplied MakeHelp compiler to create the completed help message file.

QuickHelp was developed by Harald Zoschke, Crescent Software's European distributor. The Turbo C source code is *not* available at this time.

QuickPak Scientific ($79)

QuickPak Scientific is a collection of expertly written BASIC functions and subprograms for scientists and engineers. These routines utilize state of the art algorithms, organized into seven major categories. Numerous examples and demonstrations are also included.

LINEAR ALGEBRAIC EQUATIONS

LSolve1 -	Solution of a system of linear equations.
LSolve2 -	Solution of a system of linear equations with iterative improvement.
ISolve -	Inverse of a matrix.
DSolve -	Determinant of a matrix.

ORDINARY DIFFERENTIAL EQUATIONS

DiffEQ1 - Solution of a system of first order differential equations using a fourth order Runge-Kutta method.

DiffEQ2 - Solution of a system of second order differential equations using a fourth order Nystrom method.

NUMERICAL INTEGRATION OF FUNCTIONS

Integra1 - Romberg integration of a user-defined function.

Integra2 - Integration of a function tabulated at unequal intervals.

NUMERICAL DIFFERENTIATION OF FUNCTIONS

NumDiff1 - Numerical calculation of first, second, third, and fourth derivatives of a user-defined function.

NumDiff2 - Numerical differentiation of tabulated functions.

ROOT FINDING

FindRoot - Solves for a single real root of a user-defined function. Does *not* require derivatives.

PolyRoot - Solves for the real roots of a quartic, cubic, or quadratic equation.

NSolve - Solves for the real roots of an unconstrained system of non-linear equations.

MINIMIZATION AND MAXIMIZATION OF FUNCTIONS

MiniMax1 - Finds the minima or maxima of a scalar function of one variable. Does *not* require the calculation of derivatives.

MiniMax2 - Brent's method for calculating the minima or maxima of a one-dimensional scalar function.

MultiMin - Finds the minima or maxima of an unconstrained scalar function of several variables.

CURVE FITTING

LSquares - Least squares curve fit of tabulated data.

SplineFit - Cubic spline curve fit.

TIP

One of the truly great features introduced with QuickBASIC 4 is the ability to watch a variable's value as your program works. But watching strings can be misleading. For example, if QuickBASIC shows something like:

```
YOURPROG.BAS A$:
```

The string might simply be null, but it may also consist of blank spaces. Or it could even have characters that begin past the right edge of the screen. One solution that always works is to add leading and trailing delimiters. Instead of watching only A$, instead use:

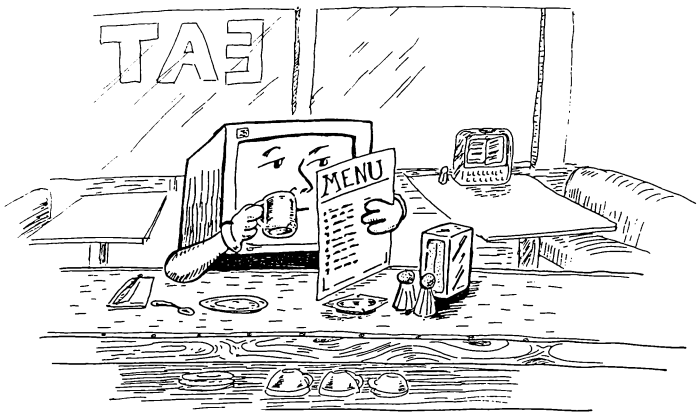
```
"(" + A$ + ")"
```

Then, if the string is full of blanks you'll see:

```
YOURPROG.BAS A$: (                    )
```

Also, because *any* legal BASIC expression may be watched, you could ask to view the 40 characters that start 100 bytes into the string:

```
MID$(A$, 100, 40)
```



QuickMenu ($59)

QuickMenu is a full-featured DOS menu system for novices and experienced users alike. Simply define the menu titles, the list of DOS commands for each choice, and they will be executed automatically. QuickMenu is simple enough for a beginner to set up and use, while providing the sophisticated options needed for the power user. All input is fully prompted, and the entire system is menu driven.

Unlike other DOS menu programs, QuickMenu does *not* remain resident in memory while it executes a program. Instead, a batch file is created and run when the choice has been selected. QuickMenu always remembers which menu level was in use, and returns there when the selected program finishes.

A file import and export feature lets you bring in existing batch files, or create new ones. QuickMenu may also be set up to prompt for additional input when a choice is selected, allowing the operator to specify a drive, file name, and so forth. Once the menus have been built, QuickMenu's *runtime only* module provides a fast, secure system that cannot be altered by the user.

PIIIIIIIIIIIIIIIIIIII

Paul Passarelli is Crescent's head of tech support, and a true math-head. Paul was telling us about a book that claims to be the "world's most boring book". Spanning 400 pages, it consists solely of the first one million digits in PI.
Here's a brief excerpt:

```
PI = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209_
       74944 59230 78164 06286 20899 86280 34825 34211 70679 82148 08651
```

THERE WILL BE A SLIGHT DELAY

Most BASIC programmers use the TIMER function in a loop to create a short delay. The only problem with TIMER is there's no way to get it to less than 1/18th second. One solution is to use the PLAY command:

```
PLAY "MFT255P64"
```

Even better is QuickPak Professional' Pause2 command. Simply tell it the number of *microseconds* you want to pause for.

BIG DEAL

QuickTALK reader Rem Outaline called the other day bragging about his new sort algorithm that's even slower than a bubble sort. Way to go, Rem.

THEN WHO NEEDS IT!

Crescent customer Tab Key wrote us to point out that BASIC's THEN keyword is often unnecessary. For example:

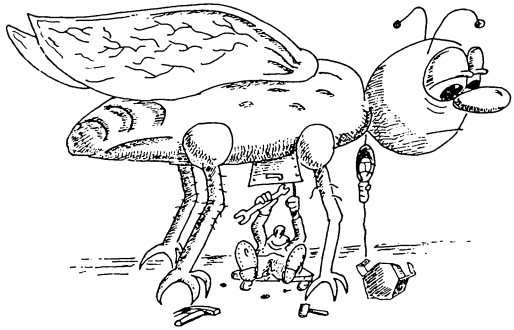
```
IF X > 2 GOTO labelname
```

Similarly, when a variable is being tested for non-zero, an explicit comparison is optional:

```
IF X <> 0 THEN . . .
```

is the same as

```
IF X THEN . . .
```



Bug Fix # 103

MICROSOFT BUGS

Microsoft is pretty good about fixing bugs and making new disks available for free. The original release of QuickBASIC 4.00 has plenty of 'em, so be sure to call them and ask for QB version 4.00*b*. The phone number for technical support at Microsoft is (206) 882-8089.

This isn't really a bug, but it could drive you up the wall if you don't know about it. In the QB environment (any version), attempting to assign an array element that is out of range will produce an error. However, in a stand-alone program that was compiled without the /D Debug option, array bounds checking is *not* performed. Therefore, if you forget to dimension an array and then assign, say, element 400, QuickBASIC will happily oblige, likely corrupting string memory in the process. If you get strange errors or a program runs incorrectly, try compiling with /D. Of course, once the program works, recompile without /D to make your program as small and fast as possible.

MASM 5.1 has a few problems as well. There's some terrific new features, but they made a major blunder—*all* procedures are now public by default. If you have a short procedure in an assembler program that is used only in that file, you can no longer have another procedure with the same name in another object module in the same .LIB or LINK library. However, Microsoft has advised us that this will be fixed in MASM version 6.

WHICH CODE IS FASTER?

by Ethan Winer

No matter what language is being considered, we all strive to create programs that execute in the least amount of time, while using the fewest number of bytes. Unfortunately, many programmers who use a high level language have no way to determine which statements are the most efficient.

Therefore, the purpose of this article is twofold–first we will discuss various BASIC statements to see which ones are faster and smaller. Then we will examine the actual machine instructions generated by QuickBASIC 4, and compare them with the original BASIC source code.

Besides seeing first hand how a compiler operates, we'll also learn about Microsoft's excellent CodeView debugger in the process. CodeView is included with a number of Microsoft languages including BASCOM 6 and current versions of the macro assembler.

All of the code examples presented here will be shown for stand-alone BASIC programs that do not require BRUN run-time library support. Also, integer variables will be used exclusively. Most programmers use integers whenever possible, and this will prevent the examples from being obscured by floating point math operations.

STRING OPERATIONS

String operations in a QuickBASIC program consist of many calls to routines contained in the BCOMxx.LIB library file. Unfortunately this makes them extremely convoluted and thus difficult to trace through. However, manipulating strings efficiently in BASIC is often more a matter of common sense than anything else.

Unlike most other high level languages, BASIC allows strings to have a varying length. Where languages such as Pascal and C require you to declare all strings and their lengths ahead of time, BASIC is much more forgiving. Of course nothing comes for free, and so it is with strings in BASIC.

Every time a string is assigned or re-assigned, a new memory location is used to hold the contents. Even if the new string is the same length or shorter than it had been previously, BASIC abandons the memory it occupied and allocates a new space in "near" memory. Near memory is where all non-array variables are stored, and is limited to 64K.

Because finding a new place to store a string takes time, this is one of the most inefficient operations BASIC does. Therefore, for our first example we'll look at two very different ways to build a string.

Let's say you intend to read a line of characters from the display screen and store them into a string. One method would be to append new characters in a loop like this:

```
S$ = ""                                 'clear S$
FOR X = 1 TO 80                         'read the top line
    S$ = S$ + CHR$(SCREEN(1, X))        'build the string
NEXT
```

Besides being slow because new memory must be found for each new version of S$, 3240 bytes of memory are left abandoned by the preceding assignments. Of course, BASIC will eventually reclaim this memory, but that takes even more time. Because BASIC leaves old string data lying around as it works, it must reorganize its string space from time to time. This process is appropriately called *garbage collection.*

Understand that variable length strings are not a bad thing. Indeed, this is just one of the many advantages BASIC has over less capable languages like C and Pascal.

LSET, RSET, AND MID$

BASIC provides two other ways to assign characters to a string–LSET and RSET, and the statement form of MID$. Where the example above must constantly assign a new string for each character that is appended, LSET, RSET, and MID$ let you insert characters into an existing string. Thus, a single string of the correct length may be assigned only once, and the new characters will be placed into subsequent locations.

Using MID$ makes the most sense for a screen reading routine, and it would be coded like this:

```
S$ = SPACE$(80)      'set aside 80 characters in S$
FOR X = 1 TO 80      'read the top line
    MID$(S$, X, 1) = CHR$(SCREEN(1, X)) 'insert each
NEXT                                     '  character
```

In a test where 5000 characters were inserted, MID$ filled the string in only 1.2 seconds, as opposed to 7 seconds for the first example.

LSET and RSET also replace characters in an existing string, however they also clear all of the characters that are not being replaced.

```
X$ = "Testing 1, 2, 3"
LSET X$ = "Hi"          'now X$ = "Hi           "
RSET X$ = "Lo"          'now X$ = "           Lo"
```

As you can see, LSET left justifies the string being inserted and RSET right justifies it.

TIP
The best way to tell if a mono or color monitor is installed:

```
DEF SEG = 0
IF PEEK(&H463) = &HB4 THEN
     'mono
ELSE
     'color
END IF
```

Another important way that string operations can be speeded up is to use their ASCII values whenever possible. As an example, suppose you need to search through a string array for a particular item. One common method would be:

```
FOR X = 1 TO ArraySize
    IF Array$(X) = Search$ THEN
       PRINT "Found at element"; X
       EXIT FOR
    END IF
NEXT
```

A better approach would be to determine the ASCII value of the string being searched for, and compare that to the value in the array:

```
A = ASC(Search$)
FOR X = 1 TO ArraySize
    IF ASC(Array$(X)) = A THEN
       IF Array$(X) = Search$ THEN
          PRINT "Found at element"; X
          EXIT FOR
       END IF
    END IF
```

Though this takes a bit more effort to code, the resultant improvement can be substantial. Because BASIC operates on integers much more quickly than it can on strings, a lot of unnecessary testing at the string level will be avoided. Even better would be to use the QuickPak Professional ASCII function. ASCII does the same thing as BASIC's ASC, but it's substantially faster.

There are certainly other similar ways that you can increase the string performance of a program. As I said earlier, many solutions present themselves once you understand how BASIC operates. On that note, let's take a look inside QuickBASIC to see some of the ways your source code is translated into executable machine instructions.

VARIABLE ASSIGNMENTS

Imagine you are writing a program, and that at some point several variables must be assigned to the same value. One way would be to assign the value repeatedly, as shown below:

```
A = 13
B = 13
C = 13
D = 13
E = 13
```

Another possibility is to assign the first variable from a constant, and the remaining variables from the first one:

```
A = 13
B = A
C = A
D = A
E = A
```

The second example is much more efficient, but let's see why.

When QuickBASIC compiles your source program, it assigns and remembers an address for each variable it encounters. This lets it replace references to those variables with addresses that the PC's processor can understand. Consider the following assignment statement:

```
A = 13
```

This is compiled to the following machine and assembler instructions:

```
C706360000D00       MOV WORD PTR [0036], 13
```

As you can see, QuickBASIC has placed the variable A at the absolute address 36, and a six-byte assembler instruction is required to assign it. (The bytes of machine language that QuickBASIC generates are shown to the left of the equivalent assembler mnemonics.) Two bytes are used for the actual instruction, another two to specify the address (0036), and two more for the value 13.

Thus, QuickBASIC will code the first example like this:

```
A = 13      C70636000D00      MOV WORD PTR [0036], 13
B = 13      C70638000D00      MOV WORD PTR [0038], 13
C = 13      C7063A000D00      MOV WORD PTR [003A], 13
D = 13      C7063C000D00      MOV WORD PTR [003C], 13
E = 13      C7063E000D00      MOV WORD PTR [003E], 13
```

Notice that two-byte memory addresses and values are always shown with the lower byte first. That is, the address 0036 is represented in memory as 3600, and the value 0013 is shown as 1300. This is not inconsistent once you think about it, because the lower byte is always stored in the lower memory location. Also notice that QuickBASIC is very organized as it assigns variable addresses, and each one occupies the next available memory location.

The second example is compiled to the following machine code, and it is immediately obvious that fewer bytes are needed. All but the first of these instructions use the value held in a register (AX), which is much faster than a constant value contained within the assembler code instructions.

```
A = 13      C70636000D00      MOV WORD PTR [0036], 13
B = A       A13600            MOV AX, WORD PTR [0036]
            A33800            MOV WORD PTR [0038],AX
C = A       A33A00            MOV WORD PTR [003A],AX
D = A       A33C00            MOV WORD PTR [003C],AX
E = A       A33E00            MOV WORD PTR [003E],AX
```

Because QuickBASIC is very smart, it creates code to retrieve the value 13 from A only once, and then uses that to assign all of the remaining variables. Therefore, the first example comprises thirty bytes of code, while the second needs only twenty one.

Another clever technique QuickBASIC performs is when incrementing and decrementing variables. The simplest type of compiler would take a statement such as X = X + 1 and create code something like this:

```
MOV  AX,[0036] ;move the value of X (address 36) into AX
ADD  AX,1      ;add 1 to AX
MOV  [0036],AX ;move AX back into X
```

But QuickBASIC is much smarter than that, and generates code to directly increment the variable X:

```
INC  [0036]    ;increment X
```

Both INC (increment) and DEC (decrement) are instructions the PC's processor can execute directly, and they are much faster than the ADD or SUB commands.

ARRAY ASSIGNMENTS

Array variables are often initialized in a FOR/NEXT loop, so let's look at that next. The examples below show how QuickBASIC codes a simple FOR/NEXT loop, as well as how variables within it are assigned. Unlike the earlier code samples, we'll show the translation similar to the way CodeView does. That is, each line of BASIC source code is followed by the compiled machine instructions.

Notice that the DIM command does not generate any assembler code because memory for a Static array is set aside when the program is compiled.

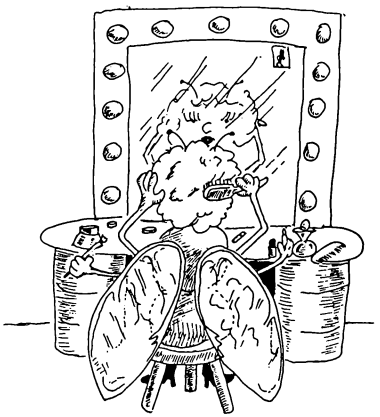
```
DIM Array(10)
FOR X = 1 TO 10
    B80100          MOV AX,0001
    E90B00          JMP Label2
Array(X) = 0
                  Label1:
    8BF0            MOV SI,AX
    D1E6            SHL SI,1
    C78436000000    MOV [SI+0036],0000
NEXT
    40              INC AX
                  Label2:
    A30001          MOV [0100],AX  (100 is X's address)
    3D6400          CMP AX,0A      (10 is 0A Hex)
    7EED            JLE Label1
```

In the example above, every element in the array is being assigned a value of zero. QuickBASIC uses the AX register

as a loop counter, and begins by initializing it to the starting count of 1. This is immediately followed by a jump into the NEXT portion of the code, where the value in AX is tested to see if the ending value of 10 (0A Hex) has been reached. If AX is Less than or Equal to 10, a Jump is made back into the body of the FOR/NEXT loop.

To determine the address of the current array element being assigned, the program moves the X count value held in AX into SI, and then multiplies SI times 2 with the SHL (Shift Left) instruction. Because each integer array element comprises two bytes, any element can be found by adding twice the current count to the address of Array(0). For example, the memory location for Array(10) is 20 bytes beyond Array(0).

The actual assignment is made by adding the computed offset held in SI to the constant 0036, which is the address of Array(0). As you can see, QuickBASIC uses the less efficient method of assigning each element from a constant value. It would have been much more efficient to first clear another register (perhaps BX) to zero, and then use the register for all of the assignments that follow.



Bug Fix # 22

SIMPLE INTEGER MATH

Of course, we shouldn't expect QuickBASIC to be as efficient as a human hand-coding in assembly language. The code created by any compiler can always be improved. However, as the short program below illustrates, when it comes to simple integer math QuickBASIC is extremely efficient.

```
X = 4
    C70636000400      MOV  WORD PTR [0036],4
X = X * X
    A13600            MOV  AX,[0036]
    F7E8              IMUL AX
    A33600            MOV  [0036],AX
X = X * 2
    D1E0              SHL  AX,1
    A33600            MOV  [0036],AX
X = X * 3
    B90300            MOV  CX,3
    F7E9              IMUL CX
    A33600            MOV  [0036],AX
X = X * 4
    D1E0              SHL  AX,1
    D1E0              SHL  AX,1
    A33600            MOV  [0036],AX
X = X + X
```

First, X (which is at address 0036) is assigned from the constant value 4. Then, AX is loaded from X, where it stays throughout the remaining assignments. To multiply X * X, QuickBASIC generates an integer multiply instruction (IMUL), and then places the result back into X.

Since QuickBASIC is smart enough to realize that AX still holds the value of X, the remaining operations are all performed on AX.

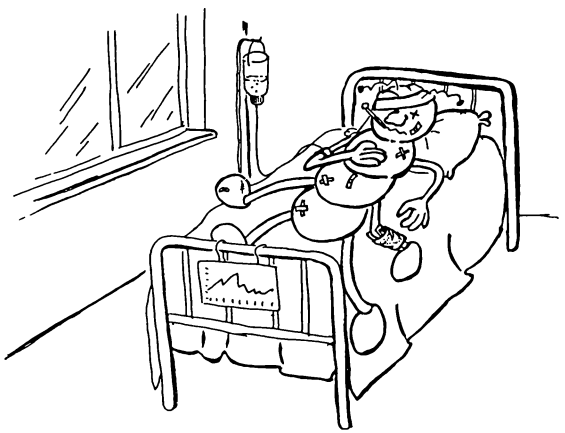
To multiply X times 2, a SHL instruction (Shift Left) is used which is much more efficient than a conventional multiply. Whenever the bits in an integer variable are shifted left one position, the value is multiplied times 2. Likewise, when the bits are shifted right, the value is divided by 2.

Multiplying times 3 cannot be done by shifting, so the CX register is first loaded with the value 3, and IMUL is then used to perform the multiplication. Once this is done, AX is moved back into X. This continues through multiplying times 4, which again is done by shifting.

Finally, the X + X assignment is handled elegantly by simply adding AX to the variable's address. Again, this code is extremely efficient because QuickBASIC is smart enough to realize that AX still holds the value of X from the previous computations.

Another important optimization that QuickBASIC performs is to calculate identical expressions only once. In the example below, A is multiplied times B only once for each pass through the FOR/NEXT loop.

```
A = 9
B = 10
FOR X = 1 TO 10
    Array1(X) = A * B
    Array2(X) = A * B
NEXT
```



Bug Fix # 8

When examined under CodeView, the program looks like this:

```
A = 9
    C70662000900      MOV  [0062],9
B = 10
    C70664006400      MOV  [0064],0A
FOR X = 1 TO 10
    B80100            MOV  AX,1
    E91900            JMP  Label2
Array1(X) = A * B
                  Label1:
    A16400          MOV  AX,[0064]  (64 is B's address)
    F72E6200        IMUL [0062]     (62 is A's address)
    8B366600        MOV  SI,[0066]  (66 is X's address)
    D1E6            SHL  SI,1
    89843600        MOV  [SI+0036],AX  (Array1(X) = AX)
Array2(X) = A * B
    89844C00        MOV  [SI+004C],AX  (Array2(X) = AX)
NEXT
    A16600          MOV  AX,[0066]  (66 is X's address)
    40              INC  AX          (AX = AX + 1)
                  Label2:
    A36600          MOV [0066],AX   (X = AX)
    3D0A00          CMP  AX,0A      (10 is 0A Hex)
    7EDF            JLE  Label1 (jump if less or equal)
```

Because neither A nor B are changed inside the loop, what QuickBASIC *should* do is calculate the result only once ahead of time. Even though many C compilers perform this level of optimization, QuickBASIC is not that smart. However, you could easily do this yourself to make the program run faster:

```
A = 9
B = 10
Temp = A * B
FOR X = 1 TO 10
    Array1(X) = Temp
    Array2(X) = Temp
NEXT
```

USING CODEVIEW

All of the examples shown thus far would have been impossible to present without the invaluable aid of Microsoft's CodeView debugger. If you are familiar with the QuickBASIC 4 editing environment, you should have little trouble understanding and using CodeView.

Like QuickBASIC 4, CodeView features break points and watch variables, and it lets you single step through a program. Also like QB4, CodeView can step over or into subprograms and CodeView is the *only* reasonable way to debug a complex assembler routine that has gone berserk. While DEBUG can be helpful for simple assembly language .COM programs, it is all but useless for debugging an assembler routine that has been added to a QuickBASIC program.

Preparing a BASIC or assembler program for debugging with CodeView is very easy—simply add the /ZI switch to the BC or MASM command:

```
BC basicprogram /ZI
```

or

```
MASM asmprogram /ZI
```

Once the program has been compiled or assembled, it must also be linked. This is done by adding the /CO switch when LINK is started:

```
LINK basicprogram [asmprogram] /CO
```

Though the logic escapes me, the LINK switch for a CodeView program is not the same as for BC and MASM. Indeed, /CO is much more sensible, and it's anybody's guess as to what /ZI is supposed to stand for.

For the most part, CodeView is not necessary if your program is written purely in BASIC. After all, the whole point of a high level language is to avoid having to deal with registers and instructions to the PC's processor. However, if you are creating assembler extensions like those produced by Crescent Software, it is invaluable. Of course, it is also the only way to see how QuickBASIC works.

USING CODEVIEW WITH QUICKBASIC

To examine the code that QuickBASIC creates, lets walk through one of the sample programs that we looked at earlier. Because we already listed the assembler code created by QuickBASIC, we'll just go over the necessary steps here. We'll get to using CodeView on a mixed program of QuickBASIC and assembler routines in a moment.

First, create a BASIC source file, preferably using one of the examples above, or a short program with just a few simple instructions. When you save it to disk, be sure to use the "Save As" option and select "Text". CodeView has no way to decipher the special "fast load" format that QuickBASIC uses as the default save method.

Next, compile the program from the DOS command line using the BC compiler and the /ZI and /O options:

```
BC basicprogram /ZI /O;
```

Once the program has been compiled, it must be linked:

```
LINK basicprogram /CO;
```

Finally, start CodeView like this:

```
CV basicprogram
```

When CodeView begins you will see a screen similar to the one used by QuickBASIC. CodeView starts up in the "immediate" mode, so you should press F6 once. At this point, you are viewing the original BASIC source instructions for your program. Pressing the F3 key lets you view the equivalent assembler instructions, but don't do that yet!

Every BASIC program begins with a large section of start-up code, which we want to skip over first. Therefore, press the F10 key to step to the first line of the BASIC source code. When CodeView is showing your BASIC source file, F10 steps to the next BASIC instruction.

Now you can press F3 to view the assembly language statements intermixed with the BASIC source. Each time F10 is pressed while you are in the assembly view mode, a single assembler instruction will be executed. Debugging a mix of BASIC and assembler source is only slightly more complicated, as we about to see.

USING CODEVIEW WITH MIXED LANGUAGES

While using CodeView with a BASIC program can certainly be an enlightening experience, it really shines when you need to follow a call from BASIC to an assembler routine and back again. As with the BASIC examples above, the assembler source must be assembled with the /ZI switch, and you must invoke LINK with the /CO option:

```
BC basicprogram /ZI /O;
MASM asmprogram /ZI;
LINK basicprogram asmprogram /CO;
```

When CodeView is started, the BASIC source will be presented first, and again you should use F10 to step to the first line in the BASIC program. When you get to a call to an assembly (or other) language routine, press F8 rather than F10. Like the QuickBASIC editor, F8 steps into a function or subroutine call, while F10 considers the call to be a single instruction and steps over it.

If the assembler source code is not in the current directory, CodeView will ask you to enter the directory name. In order to show both the source code and the machine instructions at the same time, CodeView must be able to load the original source files.

Watching assembler variables is similar to the same operation in QuickBASIC. Pressing Alt-W brings up the Watch menu, and you simply enter the variable's name. Any time the value of a watch variable changes, it will be reflected in the watch window at the top of the screen.

And that's really all their is to it! As long as you remember to use the /ZI and /CO switches, you will have no trouble tracing through either BASIC or assembly language source code. Well, that's *almost* all there is to it. There are a few bugs in CodeView that you should be aware of.

First, it is worth mentioning that there are several versions of CodeView in circulation. Second, the bugs (in the version we have) are mentioned just so you'll know it isn't your fault if you encounter them.

The first problem is that CodeView will not work if SideKick is loaded. Mind you, your program won't crash, but CodeView reports the following error:

```
run-time error R6002
- floating point not loaded
```

This error is not mentioned anywhere in the CodeView manual, and it is in fact a C language error message!

The second bug is probably QuickBASIC's fault, because it affects a TYPE variable that uses long integers. In a test program we were writing, the following TYPE declaration was placed early in the source code:

```
TYPE VLong
     Lo AS LONG
     Hi AS LONG
END TYPE
```

CodeView crashed on this one so badly that we had to turn the power off. However, the solution was simple:

```
TYPE VLong
     Lo1 AS INTEGER
     Lo2 AS INTEGER
     Hi1 AS INTEGER
     Hi2 AS INTEGER
END TYPE
```

Since we were interested in any block of data that was eight bytes, having to split each LONG into two integers wasn't a problem. Depending on what you're trying to do, a different work-around may be needed.

LEARNING MORE

We have looked at only a few samples of QuickBASIC code that have been compiled to machine instructions. Of course, many BASIC statements are difficult to follow once they have been compiled because of the complexity of QuickBASIC's system of library calls. However, tracing through some of the simpler functions can be very enlightening.

We encourage you to experiment on your own—perhaps looking at functions such as LEN and ASC—to see how these work. Indeed, our own curiosity led us to develop more efficient replacements for these commands in QuickPak Professional.

TIP

Passing a fixed-length string array or a TYPE array to a BASIC subprogram is possible, even though all of the examples in the QuickBASIC 4 manuals skirt the issue by showing the various arrays as being SHARED. The short program below illustrates the steps required to do this.

Notice that to pass an array of fixed-length strings it must be declared as a TYPE array, even if the TYPE is comprised solely of a single string member.

```
TYPE FLen                               'define the TYPE
     Stuff AS STRING * 11               '  before you refer
END TYPE                                '  to it later

DECLARE SUB FLSub (Param() AS FLen) 'declare the sub
DIM A(100) AS FLen                      'DIM the array

A(89).Stuff = "Testing #89"             'assign element 89
CALL FLSub(A())                         'call the subprogram

SUB FLSub (Array() AS FLen)             'refer to the TYPE
    PRINT Array(89).Stuff               'print the element
END SUB
```

Our *favorite* UPS driver Kevin
(actual unretouched photo).

SHIPPING, MAIL ETC...

CORRESPONDENCE WITH CRESCENT SOFTWARE

For first class please send to

Crescent Software
11 Grandview Avenue
Stamford, CT 06905

For UPS, Federal Express or anything else please call first.

Our number (203) 846-2500
FAX number (203) 849-1868

We accept MasterCard, Visa, Check, Money Orders or Cash. We do not accept open Purchase Orders, or American Express cards.

INTERNATIONAL ORDERS

Crescent Software will ship anywhere in the world (providing there is a shipper that goes there), but we also have some international dealers that might be more convenient.

In West Germany

Ingenieur - Buro
Harald Zoschke
Berliner Str. 3
D-2306 Schoenberg/Holstein
West Germany - Telefon
04344/6166

In Holland

LeMax Company B.V.
Robert Kockstraat 2
1171 BE Badhoevedorp
Holland - Telefoon 02968-
4210

In Canada

Henri Reiher
Ordinaq - Div. De L.C. Inc.
55 Cote Ste-Catherine #1604
Montreal, Quebec
Canada H2V 2A5
514-279-0309

Order Form

Name _____

Company _____

Address _____

City _____ **State** _____ **Zip** _____

Daytime telephone _____

Payment Check ( ) COD ( ) Visa ( ) MC ( )

Credit Card # _____ **Exp. Date** _____

Product	Turbo	QB2/3	QB4/BC6	Price	Total
QuickPak				79.00	
QuickPak Professional	NA			149.00	
QuickPak Scientific				79.00	
QBase				149.00	
QBase QuickScreen	NA	*		79.00	
QBase Report	*	*		79.00	
GraphPak				79.00	
GraphPak Professional				149.00	
LaserPak				79.00	
LaserPak Professional	NA			149.00	
QuickMenu				59.00	
QuickHelp				59.00	

Subtotal _____

Connecticut residents must add 7.5% sales tax _____

Shipping (see below) _____

Total _____

Shipping (per order)

U.S. $5 2nd day UPS, $10 overnight, $15 overnight for QuickPak Professional
Canada $5 1st Class, $10 QuickPak Professional (ground),
$30 QuickPak Professional (Air)/Express Mail
Europe $10 per item, $25 overnight per item, $35 for two or more
$35 QuickPak Professional express mail, overnight will be market price
Other $20 per item, $40 QuickPak Professional

* Please call for more information

Crescent Software

11 Grandview Avenue, Stamford, CT 06905 (203) 846-2500

What the people are saying!

"QuickPak is a well-rounded collection of time saving routines"
PC MAGAZINE

"QuickPak is a valuable addition to any BASIC programmer's library"
PC RESOURCE

"The manuals are written in a clear, humorous manner — a welcome change from the very serious Microsoft manuals"
TURBO TECHNIX

"QuickPak Professional includes enough features to satisfy users of all levels of BASIC expertise ... most users will regard QuickPak as a first among equals."
PC WEEK

"...Programs are FANTASTIC. They have added much excitement to my programming"
- Phoenix, AZ

"...Makes my amateur programs look professional..."
- Minneapolis, MN

"...I've come to rely heavily on Crescent Products. If I were rich, I'd buy the company"
- Schenectady, NY

"...The routines do things I could never do on my own"
- Camanche, IA

"...I have found your routines (especially the source code) invaluable..."
- Avon, CT

"I wish the ASM tutor was 300 pages!"
- Lake Zurich, IL

"Great Stuff... I may put off learning 'C' for awhile longer."
- Temperance, MI

"Found your QuickPak utilities of immense value..."
- Acton, Ontario

"...thank you for having such a great customer support and update policy."
- Chico, CA

"QuickPak has been so helpful to me, it is hard to express my gratitude..."
- Tucson, AZ

"...TOTALLY satisfied. Great work!"
- Waukegan,IL

"Your product is one of the best I have ever used."
- Lacey, WA

"You have a great product and a loyal following."
- Morristown, NJ

"Your code is beautiful and clean... your documentation is absolutely loveable!"
- Short Hills, NJ