Exploring Business BASIC

By Taylor Pohlman

The following document is produced by A.P.P.L.E. with copyright for the document based on the release of the articles contained in this book having been released into the public domain by the author:

The Third Basic

By Taylor Pohlman

A total of 23 articles about Apple III Business Basic originally published in Softalk Magazine. The articles and their accompanying programs are being presented in a series of five double-sided disks by Washington Apple Pi's III SIG, with the permission of the author.

This document was produced as a result of a necessity to have these articles in paper format for ease of access while not online. Physical production of the document was performed by Bill Martens. No claim to copyright over the material is made.

TAYLOR POHLMAN

(Originally published by TAU in its 1987
Phase III Conference Program/Seminar Guide)

Taylor has worked at Apple Computer at least twice. In his last job (in 1987 or so), he managed a group of systems engineers, product managers and technical development staff within the Business Development department.

His group supports technical requirements of programs which will help Apple get into new markets. Activities of the group include Strategic Alliances, joint development of special products for Apple's markets and other new applications of technology.

He began his career with computers as a high school teacher of math and computer science in Texas, after graduating from the University of Texas.

He subsequently held several positions in educational computing, both on the administrative side and the instructional side, culminating in managing the Instructional Computer Network, a service to public schools in northeast Texas.

In 1977 he joined Hewlett-Packard Company in education marketing and held subsequent positions in product marketing & product support, leaving in 1979 to join Apple Computer as Education Major Accounts Manager. Soon after, he became Apple][Product Marketing Manager, where he was responsible for definition of the Apple][enhancement program (][e). He then created Apple's

product support group, which had the charter for product training and third party development support. During this time, he developed Apple's third-party software licensing and technical support program. Mr. Pohlman then became Apple III Product marketing Manager in 1981, to manage the "Reintroduction" of the Apple III computer. It was at this time that he began the series "The Third Basic", a column of programming tips on Apple III Business Basic, which ran for two years in Softalk magazine.

Mr. Pohlman left Apple in late 1982 to found Forethought, Inc. and served as its first President, subsequently becoming Chairman of the Board. The company was started to focus on the "next generation" of graphic user interfaces for application software, and published four products for Macintosh, including FileMaker.

After leaving Forethought in 1985, Mr. Pohlman was co-founder and president of Regent Systems. With his partner, he designed and developed MAP, the Macintosh-based system currently used world-wide by Bank of America for internal auditing. He also managed the development of GS-BASIC, the new BASIC language interpreter for the Apple //GS (based on the Apple III's Business Basic). Additionally, he developed a Macintosh program for automation

of medical laboratory test instruments. In 1986, Pohlman sold his interest in Regent Systems and rejoined Apple.

The last we heard, Taylor Pohlman lived in Sunnyvale, California with a teenage Daughter and a cocker spaniel puppy.

NOTES ABOUT THE DISK SET

On these disks you will find all 23 articles as written by Mr. Pohlman, as well as every BUSINESS BASIC program written for use with those articles. We have placed these articles within subdirectories on each disk. For example, on this disk, the first of five, you will find on side one subdirectories for the first two articles. They are "Article1" and "Article2."

Within those subdirectories, you'll find the articles themselves, in this case named "Article.1" and Article.2." and where appropriate additional files that include the BUSINESS BASIC programs as described in the article.

The Menu.Maker program will allow you to read each article and to run the programs should you so desire. BUT PLEASE NOTE: We would recommend you copy the programs to a separate disk and run them from that disk. Since many of these programs are basically examples to show certain things that BUSINESS BASIC can do, the original program could be damaged. Further, you may need to include other files, like BGRAF.INV or other invokable module in order to properly run these programs. Where space permits, we will include these on the disk at the directory level. But you will have to use System Utilities to transfer them to the appropriate disk.

Please also note that in order to save space, we have made 3BSB-01 a self-booting disk. The rest of the Third Basic disks will include Menu.Maker but not the SOS.Kernal, SOS.Driver and SOS.Interp files needed for your Apple III to boot up. Since Menu.Maker will read and run programs from any disk placed in .D1, all you have to do is boot this disk (or any other Bootable III SIG PD BUSINESS BASIC disk) and then place the proper disk in .D1 to get your menu.

AGAIN: WE WOULD RECOMMEND YOU COPY ANY APPROPRIATE BASIC PROGRAMS TO A SEPARATE DISK AND SET THEM UP TO RUN PROPERLY FROM THERE.

FINALLY

These articles on BUSINESS BASIC are excellent and are, perhaps, the best series ever published about the language. They will also serve as an excellent beginning point for anyone interested in learning the new GS-BASIC, which was developed from BUSINESS BASIC. Many of the programs written with the III version should, with few changes, work on the GS version. GS-BASIC programs, however will not work on the III in many cases because it is much more powerful and was designed to take advantage of the GS Toolkit and other features of that machine (which are not in the III).

ARTICLES BY SUBJECT:

CONTENTS

Exploring Business BASIC, Part I

Welcome to a series of articles on Apple III Business BASIC, the powerful new cousin to Applesoft, the extended BASIC that many of you know and love on the Apple II.

My goal in this series is to make Business BASIC a useful, familiar tool for you. To do this, I'll pass along ideas that will help make the task of creating applications programs simpler and more efficient. Because Business BASIC and the Apple III itself are new to many of you, a lot of time will be spent throughout the series relating programming hints and techniques for Business BASIC to the more familiar environment of Applesoft. In fact, the ground rules of this series will be that you should be fairly familiar with the BASIC language commands and keywords, and be able to create simple programs already. Without those skills as a starting point, this series would quickly grow to be an eighty part serialization of "War and Peace".

If you are not that familiar with BASIC, your best bet is to start with the Applesoft Tutorial Manual. If you have an Apple III, simply boot the Emulation mode disk, select the Applesoft option and then insert the DOS 3.3 Master Diskette. Presto, you are now in Applesoft and can follow the Tutorial's instructions to get up to speed in BASIC. Once you are familiar with BASIC and its syntax (a word you are guaranteed to encounter in learning the language) you'll be ready to rip through the rest of these articles.

If you are already familiar with Applesoft or another BASIC, you should be ready to dig right in to Business BASIC. The series will assume that you have an Apple III in front of you in order to try out all the things that will be discussed. For those of you in that fortunate position, the fun is just starting.

Many of you have an Apple II and are wondering if you need an Apple III for that big new application or as an office complement to your Apple II at home. For you, this series should reveal the power of the Apple III, and its relationship to the Apple II. Hopefully that will help you make your decision. Others of you will just be wondering what all the fuss is about, and for you we wish happy reading. However, no matter what your situation, you should be able to gain an understanding of the power and features of Business BASIC, and maybe pick up some hints you can use in programming.

In any case, I welcome your comments, suggestions, gripes, or whatever concerning this column and Business BASIC in general. If you've written interesting routines you'd like to share, or converted programs from another variety of BASIC, or simply would like to do a core dump about your favorite subject, write to me in care of Softalk. Items of general interest will find their way into these pages, insuring immortality for both of us.

One last comment should be made, especially to those who are not business programmers. Why is Business BASIC named Business BASIC? As any product manager will tell you, dreaming up a product name ranks with dodging trolley cars and escaping from Alcatraz on the all-time "must do" list. Thus it was with Business BASIC. Certainly it's true that scientists, engineers, educators, hobbyists and lots more of you who are writing "non-business" applications will find just what you need in Business BASIC. As you stick with us in this and coming articles, however, you will see that many of Business BASIC's most powerful features were specifically designed to meet the needs of business applications, and permit the easy conversion of programs written in other business oriented BASIC dialects.

One of the other things we'll do along our way is to show how syntax in some of these other BASICs can be translated to Business BASIC. This will help you use the many reference and tutorial manuals that are on the market which use examples from other versions of BASIC. These conversion tips will also include BASIC dialects found on minicomputers and mainframes.

Well, so much for preparation. Now let's get a look at this dragon we are about to slay...

Setting the Stage

Like any other sophisticated computer system, the Apple III takes a layered approach to the operating system, languages and utilities which "animate" its hardware. By layered we mean that there are a number of levels of software which insulate the user and his application from needing to know exact details of the hardware on which his program is running.

Apple III's operating system is known as SOS (pronounced "sauce"), which stands for "Sophisticated Operating System". I know that sounds like we're bragging, but the origin of the name is interesting. Several years ago when the Apple III was under development, it was given the code name of "Sara", named after the daughter of one of its inventors. Thus SOS originally stood for "Sara Operating System". When the time came to make it an official product, the name SOS had stuck, so Marketing had to come up with another word starting with "S" that made sense. That's how Apple III's operating system became "Sophisticated". As we explore more of SOS's capabilities, you'll see that it really deserves the name.

SOS's layered approach to system control makes it more than just a disk manager (like DOS) or an I/O convention (as are IN# and PR#). SOS truly manages all of the Apple III system resources to simplify a programmer's life.

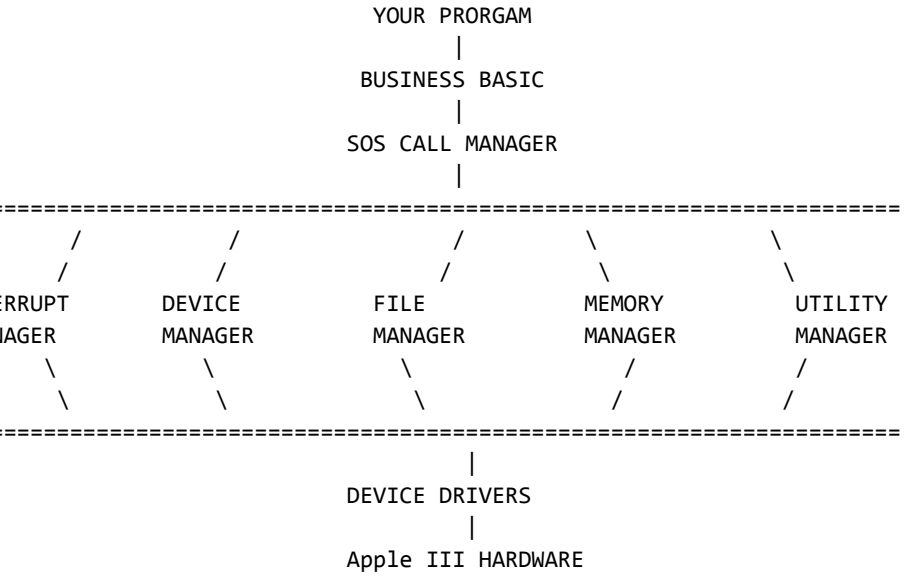
In Apple III's SOS, the lowest level of software is the hardware driver. The term "driver" may seem strange, but it's very logical. Just like the driver of a car (or a bunch of cattle) has to know the details of the operation of what's being driven,

so the Apple III drivers need specific information about how the device is connected, what its features are, how it is controlled, and what information must pass back and forth between the device and the next highest level in the system. The beauty of this scheme is that the driver can be known by some generic name (like ".PRINTER" or ".TCLOCK") so that the operating system and Business BASIC can use the device without being concerned about all the specific information that the driver must know. For example, you don't need to know anything about transmissions and turn signals to take a cab across Manhattan (a paid up insurance policy will suffice!). To extend the metaphor even further, you don't even have to know what taxi company to use, they all work pretty much the same.

In the same way, SOS can reference a ".PRINTER" for you which may be a Centronics, a Silentype, an Epson, a Qume or any number of printers, connected via parallel, joystick or serial ports. The higher you get in the operating system layers, the less specific you must be about the resources you use since SOS knows about all the devices which you have configured on your system. Facilities are also provided to allow managing devices on a demand basis (that is, when they signal that they want to do something, called an "interrupt"). This feature makes it possible to request that more than one device be active at a time. To do that on the Apple II takes some pretty sophisticated programming.

Because of the structured, layered nature of SOS, doing things like reading from a remote computer while writing a message to disk and printing out a report on a printer become almost trivial programming tasks.

We'll look at more about that later. It is sufficient to say for now that your program runs in BASIC which runs on SOS which controls the hardware drivers which accomplish the input/output to receive and deliver data for the system's devices (including a device called ".CONSOLE" which is the keyboard and screen). The structure looks something like this:



As you can see, each layer depends on the one below for services. Since the way the layers communicate is standardized on the Apple III, it is possible to make substantial changes to the hardware and even to some parts of the operating system without changing the way Business BASIC operates. This insures that your programs will continue to work, even if we make changes later. Designing operating systems this way takes longer, and makes them larger, but in the long run the benefits are enormous.

Getting Started

Since booting a disk is worth a thousand "you're gonna love it"s, let's get started by trying some things out. Just put the Business BASIC disk in the built-in drive and press Reset while holding down the CONTROL key (called "CONTROL-RESET" from here on out). The first thing you may see is a slight flicker as the onboard diagnostics check out the Apple III circuitry. Next is the SOS display screen, which indicates that the operating system has been loaded into memory. SOS's next task is to load in the language from the boot disk. Since this is the Business BASIC disk, that language is loaded and the "Hello" program is automatically run (just like DOS on the Apple II).

You'll note that the final thing to appear is the right parenthesis ")". This is the BASIC prompt, meaning that Business BASIC is ready for a command.

At this point, enjoy yourself for a minute by typing:

```
)PRINT FRE
```

You 128K Apple III owners will notice that you've got over 70K bytes of user space for programs and data. We'll find some fun things to do with all that room later! The line above also illustrates another convention we'll be using throughout these articles. What you type will always be underlined to distinguish it from what the computer outputs to you.

There are several things of interest in the display of the catalog. First, in the upper left hand corner of the printout is the name BASIC. This can vary from disk to disk and is called the "Volume Name". SOS identifies the diskette you are referencing by a scheme called the "Pathname". The highest level of the Pathname is the Volume name, with any Subdirectories mentioned next and the actual filename last (lowest) in the hierarchy. More on this subject can be found in the Apple III Owner's Guide and the Business BASIC manual under "Pathnames".

The next thing to notice is the column on file type. The type "SYSTEM" is obvious, that's SOS and BASIC, the system software. Notice that BASIC is named "SOS.INTERP", because on this diskette it is the interpreter (control program) currently configured to run on SOS. Notice also that the "BLKS" column shows the space occupied on the disk in Blocks. Blocks are 512 bytes each. The next columns, alas, alack, are only relevant to those of you who have working clock chips. The files on the Business Basic disk will have a date and time on them, but without a system clock, the files you create will not. The final column "EOF" lists the exact number of bytes occupied by the file before its "End of File" mark.

Now back to the "TYPE" column for a minute. It's easy to figure out that filetype "BASIC" stands for a basic program (like TIMESET). What does PASCOD stand for? Right, it's a Pascal code file, in this particular case created by the Pascal system's Assembler. As you might have guessed (if you've been reading your BASIC Manual), the ".INV" suffix on those files is a way of indicating that these files are set up as BASIC "Invokable Modules". We'll explore these in more detail later, but for now just remember that BASIC uses assembly language routines through a mechanism called Invoke and Perform. There are some definite rules to follow in setting up these modules which I won't go into now. However, there's no reason why we can't start using these capabilities right away! Hang on for a short exercise in using the SOS file system, and we'll give READCRT.INV a workout.

To get a glimpse of how Business BASIC works with SOS to manage system resources through files, let's take a simple example which doesn't require the disk or a printer. BASIC tells SOS that it wants to use a file by means of the "OPEN" command, and assigns a number for later reference to the file. On the Apple III of course, everything is treated as a file, even the keyboard and display. As we said earlier, the keyboard/display device is referred to as ".CONSOLE". Note that the names for all Character Devices (devices that transmit one

character at a time) start with a period. Type in the following so we can experiment (as Dr. Frankenstein said to Igor):

```
)10 OPEN#1,".console"        (this sets up a file number for
                              BASIC to use in communicating
                              to the console)
```

Note that you are already communicating to and from the console. That's because the console is the "default" I/O device. Statement 10 establishes a second path by which to communicate to the same device.

```
)20 INPUT a$                 (this is the good old ordinary
                              input to the default input device,
                              the keyboard)

)30 PRINT a$                 (again, default output device,
                              the screen)

)40 PRINT#1;a$               (now we print to the screen again,
                              this time through the console file
                              previously opened)

)50 INPUT#1;a$               (this time we input from the
                              keyboard, using the console file)

)60 PRINT a$                 (print to default screen)

)70 PRINT#1;a$               (print the same quantity to the
                              console file)

)80 END
```

Now if you LIST and RUN the result, it should look something like this:

```
)LIST
10 Open #1, ".Console"
20 Input a$
30 Print a$
40 Print #1;a$
50 Input #1;a$
60 Print a$
70 Print #1;a$
80 End

)RUN
?hello default console
```

```
hello default console
hello default console
hello console as a filehello console as a file
hello console as a file
```

A couple of interesting things are apparent here. First, although the first three lines work exactly as you would expect, the next three lines of output are a little different. The default console prints the question mark, as it should, but on line four of the output there is no question mark or prompt for input at all. This is because SOS is treating the console as a general input file and therefore can't know that it can accept characters printed to it. It just does a read to the device and waits for a end of record character (in this case a carriage return). The second unusual thing is also on line four, and that's that the PRINT command in statement 60 printed right at the end of the input string (unlike line two). The same reason applies since the carriage return you typed and the subsequent line feed the system generates for the default console are suppressed for an input file device. However, line five is printed separately, since the PRINT command in statement 60 outputs a carriage return and line feed.

In this same way, every device connected to the Apple III is available as a file. The ability to address the console devices separately will come in really handy in some future articles.

Having experimented a little with files, let's use one of those Invokable Modules we mentioned earlier and the OPEN statement to do something useful. This is a handy little utility that I use all the time to make a printout of the screen when something strange or wonderful happens.

In this example, I'm assuming that you have a Silentype as your printer. Since SOS doesn't care what device it writes to, you may substitute any output filename in line 100, even a disk text file. Try that on your Apple II!

```
)new
)100 OPEN#1,".silentype"
)110 INVOKE "readcrt.inv"
)120 FOR vertical = 1 to 23
)130 VPOS=vertical
)140 FOR horizontal = 1 to 80
)150 HPOS = horizontal
)160 PERFORM readc(@value%)
)170 PRINT#1;CHR$(value%);
)180 NEXT horizontal
)190 PRINT#1
)200 NEXT vertical
)1000 VPOS = 23 : HPOS = 1
)1010 CLOSE
)1020 END
```

Listing this program should show the following:

```
100   OPEN#1,".silentype"
110   INVOKE "readcrt.inv"
120   FOR vertical = 1 to 23
130    VPOS=vertical
140    FOR horizontal = 1 to 80
150     HPOS = horizontal
160     PERFORM readc(@value%)
170     PRINT#1;CHR$(value%);
180     NEXT horizontal
190    PRINT#1
200    NEXT vertical
1000  VPOS = 23 : HPOS = 1
1010  CLOSE
1020  END
```

Notice that this reveals another nice feature of Business BASIC. Yes, it automatically indents FOR-NEXT loops for clarity! Ever been jealous of those pretty Pascal listings? Business BASIC to the rescue!

On a more serious note, let's look at what this program does. After OPENing the appropriate file in line 100, BASIC is told to INVOKE the file "readcrt.inv".

"Readcrt.inv" is an assembly language routine which looks at the current position of the cursor. The cursor position is defined by the current values of the BASIC reserved variables HPOS and VPOS. "Readcrt.inv" then modifies the value of the variable "value%" to contain the decimal value of the ASCII character at that location. The INVOKE command tells Business BASIC to find a place for "readcrt.inv" in memory and to set up a table of all its PERFORMable routines. You can INVOKE any number of modules and BASIC will always insure that they are located in non-interfering areas of memory.

Line 120 sets up a loop which will scan the vertical lines of the screen. Line 140 sets up the inner loop which will look at every horizontal character position on that line. The routine in "readcrt.inv" is then called using the PERFORM command. Isn't this easier that a bunch of pokes and a call? Line 170 prints the character equivalent to file 1, our output file, and then takes a look at the next position. Line 190 makes sure we print a carriage return at the end of each output line (since that character isn't physically on the screen). After that, line

200 starts scanning the next line. Line 1000 through 1020 set the cursor to the bottom of the screen, close the output file and end.

Now for the fun. Run this program and you will get an exact copy of the first 23 lines of the screen on your output file. By putting in an INPUT statement to ask for the filename and then OPENing the resultant string variable as the filename in line 100, you can decide at the time you run where you want the copy to go. Use this program to document all the strange and wonderful things you find in Business BASIC as you begin to really explore the language. But first, be sure to SAVE the program to an initialized diskette

Well, that's it for now. Until next time, happy coding with the most powerful BASIC around!

Exploring Business BASIC, Part II

When last we left the Lone Ranger, huge boulders crashed down the slope toward his tiny campfire... No, I'm sorry to say that the September column wasn't quite that breath-taking or cliff-hanging. However, taking the Apple III out for a spin does excite a lot of people, and I hope that includes you. If you haven't read last month's column, I recommend you get a copy. This series is progressive in that each article builds on the previous one. I'm going to assume that you have been following the series, so that each month we can cover a new topic in the least possible amount of my purple prose.

Another note before we start: as I mentioned last time, I welcome questions and comments on this series of articles or BASIC in general. Due to deadlines (Margo's Bane, we call them), I am writing this article before the previous month's is in print. Thus my reaction to your timely comment will be somewhat delayed. Let those cards and letters roll in and the responses will show up just as soon as inhumanly possible.

The SOS File System Revisited

After a brief discussion of the Apple III SOS file system last time, I concluded with somewhat of a challenge for you. We were working with a program to dump the contents of the screen to the Silentype printer, and I mentioned that the program could be generalized for any file, including disk text files. The point was that SOS took care of all the details about how each device worked, so that the user could change things at will. For reference, here's the program with which we were working:

```
50 OPEN#1,".silentype"
90 INVOKE "readcrt.inv"
150 FOR vertical = 1 to 23
155 VPOS=vertical
160 FOR horizontal = 1 to 80
165 HPOS = horizontal
170 PERFORM readc(@value%)
180 PRINT#1;CHR$(value%);
190 NEXT horizontal
200 PRINT#1
210 NEXT vertical
900 VPOS = 23 : HPOS = 1
1000 END
```

Before we modify this program to generalize it, did you try to simplify the program by directly using VPOS and HPOS in lines 150 and 160? By that I mean:

```
150  FOR VPOS=1 TO 23
160  FOR HPOS=1 TO 80
```

If you do, BASIC responds with the classically familiar "SYNTAX ERROR", since VPOS and HPOS are "Reserved Words" and cannot be used as index variables.

To continue, the challenge was to generalize the screen dumping program so that the output could go to any file. Here's one solution to that problem:

```
50  VPOS=23:HPOS=1
60  INPUT"Name of file to dump screen to: ";filename$
100 Open#1,filename$
110 Invoke "readcrt.inv"
120 For vertical =1 to 23
130 VPOS=vertical
140 For horizontal = 1 to 80
150 HPOS=horizontal
160 PERFORM readc(@value%)
170 PRINT#1;CHR$(value%);
180 NEXT horizontal
190 PRINT#1
200 NEXT vertical
210 CLOSE
300 VPOS=23:HPOS=1
310 END
```

Several differences are worthy of note. First, the cursor has been repositioned in line 50 to the bottom of the screen to avoid overwriting any existing data. The user is then prompted in line 60 to input the name of the output file. Note that this can be any legal filename on the Apple III which accepts output (printers, the communications port, a disk text file, even .CONSOLE itself)

Note also the addition of the CLOSE statement at line 210. This insures that all files are properly written to and dispensed with at the conclusion of the program. Failure to properly close files can leave some data still in memory (since files aren't automatically closed at the end of the program). This can have some interesting consequences if the file in question is a disk file and you switch to another diskette which does not have that file created on it. Now is the time to form the habit of closing all files at the end of a program.

Running this program can be instructive. Obviously, if you reply ".SILENTYPE" to the prompt, it will work like the first example. Now try replying ".CONSOLE". After the usual initial whirring of the disk to load the Invokable Module, the program appears to go to sleep for 40 seconds or so. What's happening is that

the program is reading a character and then copying it back on top of itself! The Apple III is working its little heart out and the result is as exciting as watching bread mold. Now try replying with a disk file name (you can just make up a name, as long as it follows the filename rules). The disk will whir as before. This time it has two jobs to do. First, it must OPEN the disk file using the name you gave it (let's assume you typed MYFILE.SCREEN). This means creating an entry in the directory of the current disk volume, finding initial space for the file, and setting up a "buffer" area in memory for communication of data to and from the file. Since Apple III divides the disk up into "Blocks" of 512 characters, this internal buffer is 512 bytes. This buffer size if fixed no matter what record size you specify. Later on in this article we'll look at techniques which use that piece of information to insure maximum efficiency and performance in disk-based application programs.

Once the file is opened, BASIC then INVOKES the READCRT module, and execution begins. Notice that although the printer in our previous example started almost immediately, there is a noticeable pause before the disk spins into action, and it appears to spin only four times before the program stops.

What's happening is this: line 170 prints one character at a time into the buffer. After 80 characters, line 190 prints a carriage return into the buffer and then starts the next line. After a little over 6 lines of the screen (480 bytes plus 6 RETURNs plus 26 bytes of line 7 to be exact) the 512 byte buffer is full and must be written to disk. That's the first spin of the disk which writes block 1 of the file. Next, the block number is incremented, and more writing starts from line 7 of the screen. 512 bytes later the same process is repeated until all the screen is read by the program and written into the last buffer. Some arithmetic would convince you that BASIC is in the middle of its fourth buffer when the program finishes reading line 23 of the screen. Thats when the previous comment about being sure to close files comes in handy. The CLOSE command in line 210 forces the current buffer to be written to disk, even if its not full, and the directory entry is updated to reflect the new file information.

After running the program, the CATALOG listing of the file should look something like this:

```
TEXT   00005 MYFILE.SCREEN    00/00/00 00:00 00/00/00 00:00  1863
```

Notice that BASIC identified the file as a TEXT file automatically, because the PRINT# command was used to write to it. Notice also that the Blocks Used column shows four. That disagrees with what we had predicted, since the screen data should have been able to fit into 4 blocks (2048 bytes). The reason for the extra block is that SOS allocates an extra block as an "index" block, to store information about where the rest of the blocks in the file are physically located. This insures that a large file can be created, even if the disk is fragmented into small areas of unused space. If you look closely at the directories of various files, you will note that all of them have one more block

than the EOF column would indicate, except for the one block files, which have no need of an index block. In this case, the EOF (End Of File) is after 1863 bytes. That works out to 23 lines of 80 characters (1840 bytes) plus 23 carriage returns for a total of 1863 bytes. "Close enough for folk music", as they used to say at my high school!

One last subject before we move on to further explore files. The Silentype gave us a permanent record of what was on the screen, but since this time we wrote the results to a disk file, we need a way to dump the contents of MYFILE.SCREEN to the printer. The following program easily accomplishes the task and serves as a general file to file transfer program:

```
5  INPUT "Name of file to dump: ";inputfile$
10 OPEN #1,Inputfile$
15 console=0
20 INPUT "FIle to dump to: ";outputfile$
25 OPEN #2, outputfile$
30 check$=MID$(outputfile$,1,3)
35 IF check$=".co" OR check$=".CO" OR check$=".Co" THEN console=1
40 IF console then HOME
45 ON EOF#1 GOTO 65
50 INPUT#1, a$
55 PRINT#2, a$
60 GOTO 50
65 IF console THEN HPOS=1:VPOS=23
70 CLOSE
75 END
```

There. As long as you don't try to read from the printer and print to the keyboard, it should work fine!

Note also that we have checked in line 35 to see if the device being written to is .CONSOLE. If so, line 40 clears the screen to reproduce exactly what was there when the original program was run. Line 65 repositions the cursor to the bottom of the screen so that the prompt will not cause the top line to scroll out of view.

More on Files

The subtle and nefarious purpose of this lesson, if you haven't realized by now, is to provide more insight into Business BASIC disk files. Unlike tutorials, which by now would have gotten to the mysteries of FOR-NEXT loops, I have remained true to the promise of the first article and assumed that you have some skill in BASIC. Bear with me as things get more interesting...

So far we have considered only the type of disk files which are referred to as "Text" files. These are files which contain ASCII characters which are

representative of what would be printed out if we wrote data to the screen instead of disk. For now we'll stick with this file type and later touch on "Data" files, a quasi-unique file type on the Apple III.

We have already discussed the fact that the disk is organized into 512 byte blocks. In fact, BASIC text file records can be of any reasonable size. Instead of using the OPEN statement which assigns a default of 512 bytes, we could have used the CREATE statement which allows up to 32767 byte records to be used. Of course, the record size of a particular file is of no consequence if we are merely going to read each string in order (as we did with the contents of the screen). The real power of creating files of various record sizes is to be able to randomly read data on a particular item in the file without having to deal with the other data in the file. For example, if we had wanted to print the twenty first line of the screen in the previous example, it would be necessary to input the first twenty lines, discarding the data, and then finally read and print the line we wanted. A much more efficient way would be to create the file as a random access file with record size 81 bytes. Since each record will correspond with one line of the screen, we have an easy way to address the data in question. Compare the examples below with the previous sequential access examples:

```
50  VPOS=23:HPOS=1
60  INPUT"Name of file to dump screen to: ";filename$
70  CREATE filename$,TEXT,81
100 OPEN#1,filename$
110 INVOKE"readcrt.inv"
115 cum$=""
120 FOR vertical=1 to 23
130   VPOS=vertical
140    FOR horizontal = 1 to 80
150    HPOS=Horizontal
160    PERFORM readc(@value%)
170    cum$=cum$+CHR$(value%)
180    NEXT horizontal
190   PRINT#1,vertical;cum$
195   cum$=""
200   NEXT vertical
210   CLOSE
300   VPOS=23:HPOS=1
310   END
```

Note that we have added line 70 to create the filename with the proper record size. The notation of "TEXT" is extra baggage, since the PRINT statements in the program will automatically define it as a text file, but it is good practice to be specific. I have also added a new wrinkle in lines 115, 170, 190 and 195. Instead of printing each character as it is read, the variable "cum$" is used to accumulate characters as they are read from the screen. Line 190 prints the

entire line of the screen using the vertical position as the record number. The result when running this program seems the same as when running the sequential version, except for one thing. If you CATALOG the resulting filename, it should look something like this (assuming a name of SCR.DUMP.RND)

```
TEXT    00005 SCR.DUMP.RND    00/00/00 00:00 00/00/00 00:00  1944
```

Everything is the same except the length! It turns out that when a file is created that the first record is record 0, not record 1. This is consistant with the first element of an array being element 0. Therefore BASIC has reserved 24 (not 23) records of 81 bytes each for a total of 1944 bytes.

Now that we have associated a record number with every line on the original screen, we can locate a given line by just giving its number, instead of having to read through all the other lines to find it. Witness the modified read program below:

```
5  INPUT "Name of file to dump: ";inputfile$
10 OPEN #1,Inputfile$
15 console=0
20 INPUT "FIle to dump to: ";outputfile$
25 OPEN #2, outputfile$
30 check$=MID$(outputfile$,1,3)
35 IF check$=".co" OR check$=".CO" OR check$=".Co" THEN console=1
40 IF console then HOME
45 ON EOF#1 GOTO 65
47 INPUT"record number to dump: ";rec
48 IF rec=0 THEN 65
50 INPUT#1,rec;a$
55 PRINT#2,rec;a$
60 GOTO 47
65 IF console THEN HPOS=1:VPOS=23
70 CLOSE
75 END
```

This program is very similar to the previous dump program except that line 47 asks for the specific record to dump, line 48 gives us a way out by checking for 0 and line 50 has been modified to read directly to the record number which was previously entered.

Some experimentation with this program will produce interesting results. Try reading record 1, 6, 12 and 18. In each case you will cause a disk access (the whirring sound is a clue) to read the particular record. Now try reading record 6,7,8,9 and 10 in any order you choose. The first record you read will probably cause a disk access, but the others should occur virtually instantaneously without causing disk activity. This is because SOS is still buffering files in 512 byte blocks, and all those records fall within one block. There was no need to re-read the disk since the data was already in memory. Careful planning of your

record sizes and reading sequences can have the effect of substantially increasing the performance of your program if as many reads as possible occur within the current buffer.

One interesting postscript before we proceed: If you ask for record 6 you will typically trigger a disk read as we have said. If you immediately request record 5, another disk read will be performed. This is what you might expect, but more is going on here than meets the eye. Simple calculation will prove that record 6 actually occupies space in both block 1 and block 2 of the file. The first six records, 0 through 5 occupy 6*81 or 486 bytes of the first block, leaving only 26 bytes left in that first block for record number 6. The remaining 55 bytes are in block 2. Thus a read to record 6 actually triggers two disk reads, one for the complete block 1 and one for block 2. A little more arithmetic will show which other records are in this same situation. The moral is simple: if possible, make your record sizes such that they evenly divide into 512 or are a multiple of 512. That may waste a little space, but may be more than made up for in the ability to predict when disk access will take place.

A Final Challenge

I just reviewed the last five or six paragraphs and discovered that my usual humorous style has been replaced by long detailed discourses of unrelieved tedium. There is, unfortunately, no let up in sight.

To this point we have been using "record number" files (called "random access" by most people) with record numbers which span a rather narrow range. SOS permits random files to have record numbers in the range of 0 to 32767. However, SOS does not demand that a file actually have all the records present on disk. Records are allocated as written, with only a little space taken up to keep track of where everything is. To illustrate the power this gives, consider the following problem:

A distribution company wants to keep track of their part numbers and descriptions. The part numbers are four digit numbers. Listed below is a simple program to create the part number file. Between now and next time, you could try writing a program to retrieve part number information randomly, and make changes as required. Without further ado...

```
5    HOME
10   PRINT"Parts file Create and Add program"
20   PRINT
30   PRINT"Type 1 to Create a parts file":PRINT
40   PRINT"Type 2 to add to an existing parts file."
50   PRINT:INPUT"Your selection: ";a$
60   IF a$="" THEN 1000
70   a=VAL(a$)
```

```
80   ON a GOTO 100,400
90   GOTO 5
100  PRINT:INPUT"name of new parts file: ";a$
110  IF a$="" THEN 5
120  CREATE a$,TEXT,64
130  PRINT"Parts file ";a$;"created."
140  GOTO 5
400  PRINT:INPUT"Name of existing parts file: ";a$
410  IF a$="" THEN 5
420  OPEN #1,a$
430  HOME
500  PRINT:INPUT"Part number to add: ";a$
510  IF a$="" THEN 5
520  a=VAL(a$)
530  IF a<1 OR a>32767 OR INT9a)<>a THEN 500
535  rec=a
540  rec$=a$+"\"
545  PRINT:INPUT"Description: ";a$
550  IF LEN(a$)>30 THEN a$=MID$(a$1,30)
560  rec$=rec$+a$+"\"
570  PRINT:INPUT"Location: ";a$
580  IN LEN(a$)>10 THEN a$=MID$(a$,1,10)
590  rec$=rec$+a$+"\"
600  PRINT: INPUT"Quantity on hand ";a$
610  a=0:a=VAL(a$):IF INT(a)<>a THEN 600
620  rec$=rec$+a$+"\"
630  PRINT:PRINT"Record is: ";rec$;" OK?"
640  INPUT"";a$
650  a$=MID$(a$,1,1):IF a$<>"y" and a$<>"Y" THEN 430
660  PRINT#1,rec;rec$
670  PRINT:PRINT"Record added."
680  GOTO 430
1000 PRINT:PRINT"End of parts file program."
1010 CLOSE
1020 END
```

This does not presume to be a model program in terms of its error checking, efficiency or even logic design (note all the GOTO's, patently offensive to the initiated). I tried to keep the program simple and straight-forward, allowing plenty of room for improvements. One or two things are worth pointing out to help you with your inquiry program. Since each field could be of variable length within certain maximums, I used the backslash character to delimit each item. You'll want to strip these out when you retrieve the record. Look up the function INSTR, it'll make it easy.

Once you've typed this program in, trying it out can be interesting. Try several values for part number, including some larger ones (greater than a thousand at least). Unless you add records which are sequential, each one will probably trigger a disk access as the appropriate block is written to disk. After adding several, get out of the program by typing RETURN to the part number and selection prompts and check out the catalog entry on the file. Assuming you used the name MY.PARTS as a file name when you used the create option, the entry will look something like this:

```
TEXT   00007 MY.PARTS    00/00/00 00:00 00/00/00 00:00  85735
```

Look at that EOF value! It seems that you have a huge file until you notice that the Blocks Used column is still pretty small. What SOS has done is report the EOF at the end of the highest record number you used, while allocating only those blocks which it actually needed. Some micros (and some mainframes for that matter) would require that all the blocks be all allocated before any could be written.

Well, have fun until next time. Then I'll try to lighten it up a little as we talk about the mysterious DATA file type and start using the massive amount of memory in the Apple III for some really fast indexing schemes. Before this is over you should be able to write some pretty hot database programs. 'Till then ponder the following: Is it true that disk-based programs are written by BLOCKheads?

Exploring Business BASIC, Part III

Lots has happened to the Apple III since my last article, and I appreciate all your comments about the articles in this series. We'll have a chance to pick up on some of your suggestions next month, along with more news about the Apple III. For now we'll continue our exploration of the Business Basic file system as promised last time. After reading this article and working with the examples, you should have a good knowledge of the differences between the TEXT and DATA file types as well as more information about string handling functions and techniques. We are going to stick with relatively simple indexing techniques for now, but next month we'll also cover some advanced indexing and file access methods to give you an idea of some of the ways that the popular "Data Base" programs retrieve data so rapidly.

Looking Back

The last article concluded with an example program that showed how the SOS file system could be used to store and rapidly find records in a file. We did that by using the random file access method that SOS and Business Basic have built in. That technique allows file records to be numbered from 0 to 32767 and read directly without having to read all records from the beginning of the file. The example I gave at the end of last month's article also demonstrated that SOS uses a special storage and indexing method which wastes very little space in storing records on the disk, even if they have widely different record numbers.

To go into further depth on this subject, and to compare the TEXT file type we were working with last time to the more mysterious DATA file type, it's going to be necessary to create a more general version of last month's program. Last month's example program allowed us to create a file which contained four pieces of information about a hypothetical parts distribution company. This information was the part number, a description of the part, the location where the part was stored, and the quantity on hand. Unfortunately, the program was just designed to make some clever points about files, not to really be useful to parts companies. For example, we could only perform two functions, creating files and adding records. Most parts companies would want to look up parts, delete parts, get lists of parts, etc. This month's version gets closer to that ideal, without denying you some of the fun of making your own changes. In addition, I have generalized some of the functions that the program performs into subroutines, so that we can make changes later without wholesale rewriting.

The new Parts Program

Well, now that you're breathless with excitement, here's the new version of the program:

```
5   HOME
7   PRINT
10   PRINT"Parts File Create and Modify Program"
20   PRINT:PRINT"Type:"
30   PRINT"     1 to Create a parts file":PRINT
40   PRINT"     2 to Use an existing parts file":PRINT
49   PRINT"     9 to Quit":PRINT
50   PRINT:INPUT"Your selection: ";a$
60   IF a$="" THEN 1000
70   a=ABS(VAL(a$))
80   ON a GOSUB 100,400
90   IF a=9 THEN 1000:ELSE 5
100   PRINT:INPUT"Name of new parts file: ";a$
110   IF a$="" THEN RETURN
120   CREATE a$, TEXT,64
130   PRINT"Parts file ";a$;" created."
140   RETURN
400   HOME
405   PRINT:INPUT"Name of existing parts file: ";a$
410   IF a$="" THEN RETURN
420   OPEN#1,a$
425   file$=a$
430   HOME
435   PRINT:PRINT"Modify Parts File ";CHR$(34);file$;CHR$(34):PRINT
437   PRINT"Type:"
440   PRINT"     1 to add to your parts file":PRINT
445   PRINT"     2 to delete a part from your parts file":PRINT
450   PRINT"     3 to find a part in your parts file":PRINT
460   PRINT"     9 to quit the modify mode":PRINT
465   PRINT:INPUT"Your selection: ";a$
467   a=ABS(VAL(a$))
470   IF a=9 OR a$="" THEN RETURN
475   ON a GOSUB 500,700,800
480   GOTO 430
500   HOME
505   PRINT:INPUT"Part number to add: ";a$
510   IF a$="" THEN RETURN
520   a=VAL(a$)
530   IF a<1 OR a>32767 OR INT(a)<>a THEN 500
```

```
 535   rec=a
 540   partnum$=a$
 545   PRINT:INPUT"Description: ";a$
 550   IF LEN(a$)>35 THEN a$=MID$(a$,1,35)
 560   desc$=a$
 570   PRINT:INPUT"Location: ";a$
 580   IF LEN(a$)>15 THEN a$=MID$(a$,1,15)
 590   location$=a$
 600   PRINT:INPUT"Quantity on hand: ";a$
 610   a=0:a=VAL(a$):IF INT(a)<>a OR a>99999 THEN 600
 620   quantity$=a$
 630   PRINT:PRINT"Record is:
";partnum$;"/";desc$;"/";location$;"/";quantity$;
 640   INPUT"  Ok? ";a$
 650   a$=MID$(a$,1,1):IF a$<>"y" AND a$<>"Y" THEN 505
 660   GOSUB 2000
 665   IF errorcode=0 THEN PRINT:PRINT"Record added.":GOSUB 995:GOTO 500
 670   PRINT:INVERSE:PRINT"Record not added, ERROR= ";:NORMAL:PRINT
errorcode:G
       OSUB 995:GOTO 505
 700   HOME
 705   PRINT:INPUT"Part number to Delete: ";a$
 710   IF a$="" THEN RETURN
 715   a=VAL(a$)
 720   IF a<1 OR a>32767 THEN 700
 725   rec=a
 730   GOSUB 1800
 735   IF errorcode=1 THEN PRINT:PRINT CHR$(7);"No such part
number":GOSUB
995:GOTO 700
 740   PRINT"Delete ";partnum$;"/";desc$;"/";location$;"/";quantity$;"?
";
 745   INPUT"";a$:a$=MID$(a$,1,1)
 750   IF a$<>"y" AND a$<>"Y" THEN PRINT"Not deleted":GOSUB 995:GOTO 700
 755   GOSUB 1900
 760   PRINT:PRINT CHR$(7);CHR$(7);"Record deleted":GOSUB 995:GOTO 700
 800   HOME:PRINT
 805   INPUT"Part number to find: ";a$
 810   IF a$="" THEN RETURN
 815   a=VAL(a$)
 820   IF a<1 OR a>32767 OR INT(a)<>a THEN 800
 825   rec=a
 830   GOSUB 1800
 840   IF errorcode=1 THEN PRINT:PRINT"No such part number":GOSUB
995:GOTO 800
```

```
850   PRINT:PRINT"Part number:      ";partnum$
855   PRINT:PRINT"Description:      ";desc$
860   PRINT:PRINT"Location:         ";location$
865   PRINT:PRINT"Quantity on hand: ";quantity$
870   PRINT
890   PRINT:INPUT"Press RETURN to continue: ";a$:GOTO 800
899   REM
900   REM delay subroutine
901   REM
995   FOR i=1 TO 1000:NEXT i:RETURN
996   REM
1000   PRINT:PRINT"End of parts file program."
1010   CLOSE
1020   END
1799   REM
1800   REM retrieve a record with record number = "rec"
1801   REM
1805   errorcode=1
1810   ON EOF#1 RETURN
1820   DEF FN scan(start)=INSTR(rec$,"/",start)-start
1830   INPUT#1,rec;rec$
1835   IF rec$="" THEN RETURN
1840   pointer=1:length= FN scan(pointer)
1850   partnum$=MID$(rec$,pointer,length)
1855   pointer=pointer+length+1:length= FN scan(pointer)
1857   Desc$=MID$(rec$,pointer,length)
1860   pointer=pointer+length+1:length= FN scan(pointer)
1870   Location$=MID$(rec$,pointer,length)
1875   pointer=pointer+length+1:length= FN scan(pointer)
1885   Quantity$=MID$(rec$,pointer,length)
1890   errorcode=0:RETURN
1899   REM
1900   REM delete a record with record number = "rec"
1901   REM
1905   PRINT#1,rec;""
1910   RETURN
1999   REM
2000   REM add a record with record number = "rec"
2001   REM
2005   errorcode=0
2010   rec$=partnum$+"/"+desc$+"/"+location$+"/"+quantity$+"/"
2015   ON ERR GOTO 2040
2020   PRINT#1,rec;rec$
2030   OFF ERR:RETURN
```

```
 2040   errorcode= ERR:OFF ERR:RETURN
```

Well, nobody said that this series wouldn't get more interesting as we went along! Let's take a quick look at the changes in this version of the program, as well as its major features.

First, as to structure, the program looks something like this:

```
5-90       Initialization and first menu
100-140    Create a new parts file
400-480    Open an existing file and set up second menu
500-670    Add a record
700-760    Delete a record
800-890    Find and display a record
900-995    Subroutine to create a delay
1000-1020  Terminate the program and close files
1800-1890  Subroutine to find a record and load data values
1900-1910  Subroutine to physically delete a record
2000-2040  Subroutine to add a record with given data values
```

Note that for simplicity we have assumed a fixed file record structure. That is, we have "hard coded" into the program the fact that the data items in each record are Part number, Description, Location, and Quantity on hand. We have also coded into the program some restrictions as to the length of each item (lines 550,560 and 610). A real "data base" program would have all this information stored in tables for more flexibility. For example, there is no practical way, short of rewriting parts of the program, to add an extra data item to the records, or to change the meaning of the existing items. Obviously, the more such generalizations we put in the program, the larger and more complex it will be. Our purpose is to learn something about files first, and then to write the world's greatest database program.

To help understand the program and to also check out a few new features that make Business Basic really handy, let's look at the subroutines in the program.

First, examine the record retrieval routine at line 1800, which is used by the "Find" section and the "Delete" section. We will communicate any problems encountered in a subroutine by using the "errorcode" variable, with 0 indicating no error found. The ON EOF statement in line 1810 will return with "errorcode" set to 1 in the event that the INPUT statement in line 1830 reads past the current end of file. Line 1820 sets up a function definition that comes in pretty handy. The function "scan" uses the Basic INSTR function to determine how many characters there are to the next occurrence of the "/" character. Remember that we used the "/" character to delimit the fields within the string record we stored in the file. The INSTR function returns the character position of the string being searched for, starting with the position given by "start". Subtracting the

starting value from the position gives the total length of the field. More about INSTR can be found in the Business Basic manual. Give that section a look, because INSTR is one of the most useful functions you'll find. Some other Basics may use a different name for this function (POS is one example I know of).

Line 1830 inputs the record according to the record number "rec". After checking for a "null" string (line 1835), lines 1840 through 1885 are responsible for breaking up the record into its separate fields. This is done by setting the variable "pointer" to the beginning of the field and then setting length to the number of characters in the field using the "scan" function defined previously. The MID$ function then is used to make the assignment to the appropriate variable. Study this section carefully to be sure you see how this works. My technique to understand routines like this is to make a diagram of the data and work through the statements while "playing computer".

Now that the individual fields are assigned to the proper variable, they can be used in the calling routines (at lines 730 and 830) to display the values as desired. Later on we are going to change the structure of this file considerably, and it will be handy to be able to handle that by changing the routine at 1800 rather than making changes throughout the program.

The delete routine at line 1900 is really simple, just consisting of printing a "null" record over a previously existing record. As we change the file structure, this may well become much more interesting.

The add routine is also simple, consisting (at line 2010) of packing the various field values together using the String Concatenation Operator (the world's longest way to refer to "+"). There is one thing of interest, however. Note that the ON ERR statement is used to trap any errors which may occur in writing to the file. We again use "errorcode" to communicate that an error has occurred, and are careful to turn error trapping off before returning to the main routine. It would have been possible and even desirable to use the ON ERR statement to check for all errors in the program, but the routine required to make the program that "bulletproof" would have made this program unnecessarily long. That's probably a good subject for a future article.

Well, now that we've been through the major features of the program, I suggest that you enter the program and start fooling around. As I mentioned last time, this program was never meant to be the ultimate in user friendliness or elegance of coding style. As you add records, then find and delete them, try to imagine ways you could improve the way the program works, or how it asks for information.

Business Basic "DATA" files

While it is certainly true that most files contain data, Business Basic uses the term "DATA files" in a special way. You may remember that a TEXT file consists of strings of characters with the "carriage return" character as the terminator between strings. If you print a numeric variable into a text file, it will automatically be converted to a string value, just as is done when printing numbers to the screen. This sounds pretty nice, but it can cause some inconvenience and some real problems. For example, you know that an integer variable (which ends with the "%" character) occupies two bytes of storage in memory. However, representing the value in string format can take up to six bytes (-32000 for example). Add a RETURN character to delimit it and you have up to seven bytes to store an integer in a file. Furthermore, it is not possible to tell beforehand just how much space a given set of numeric variables will take, without checking each one beforehand. This can cause some design problems for programmers. As you can imagine, these problems are even more acute for the "long integer" data type, which can be up to nineteen digits long, but only requires 8 bytes of internal storage.

There's another problem with using text files which only shows up when using "real" numbers. Reals are represented in Business Basic as 32 bit floating point quantities requiring four bytes of internal storage. Normally they are displayed with six digits of precision, and the format itself may vary greatly, especially if the magnitude of the number is very large or very small. In those circumstances, Basic will display the number in "scientific notation". This means that the output format of a "real" can vary from something simple like "3.45" to something like "-1.36723E-06". Interestingly enough, it is not so much the space that this notation takes up that causes the trouble, but the fact that the printed representation of a "real" may not correspond exactly to the value stored in memory. If the representation of a number is not exact or requires more decimal places than can be displayed, then the number is rounded before printing. By contrast, this does not occur with integers. Since rounding occurs during printing, and text files are storage of the printed format, values of real numbers may be different in the text file than they were in memory. A short example will illustrate:

```
10   OPEN#1,"numberfile"
20   INPUT"type two numbers: ";x,y
30   z=x*y
40   PRINT#1,1;z
50   INPUT#1,1;z1
60   IF z=z1 THEN PRINT"they compare":GOTO 20
70   PRINT"they don't compare: ";z,z1
80   GOTO 20
```

Note that by printing the value to the file with the random access method in line 40, we are able to read it back directly in line 50. This lets us check to see if any value change has occurred as a result of the file operation. Try this with values like 500 and 4.25. Everything should go normally. Now try a value like 3.033 and .031. Still ok. Now try 3.031 and .031. The result should print out appearing exactly the same, yet the comparison in line 60 fails. If you wish you can insert a statement at line 75 to print out the difference. It will be small, but obviously significant. For the real reason why the product of this pair of numbers fails to work, I commend you to your local math professor or textbook on numerical analysis. Suffice it to say that certain real numbers cannot be stored exactly as binary numbers nor can certain binary numbers be displayed exactly in a finite number of digits. As soon as these situations occur, the quantities stored in the text file will not exactly match what was calculated in memory. Play around with this program further. There are nearly infinite numbers of combinations which will also fail the test, but appear to be equal.

You've just seen two reasons why there is a need from time to time to store numbers in a file in the exact form which they have in memory. Can you think of a circumstance where you might want to do that with a string? Right! Among others, if you have a string which contains (or could contain) a RETURN character, the text file input statement will terminate wherever the RETURN occurs, thereby losing the rest of the characters in the string.

The key is that with DATA file format, you can store any numeric or string quantity without worrying about what might happen to the information. In addition, Business Basic adds an identifier to the front of each item of data you store in a DATA file, to indicate what kind of data it is. This is called the data Type, and allows you to intermix integers, reals and strings in any order and still read them back correctly. The information about the type of a particular data item is retrieved, astoundingly enough, by the TYPe function. This allows a simple program to read the contents of any DATA file, without having any information about it beforehand. Much more information about DATA files can be found in your manual, and I suggest you spend some time reviewing it. In the meantime, lets look at a what using DATA files will do to the parts program I listed at the beginning of the article.

First, we'll need to change the file type specification on the CREATE statement at line 120. The new line will look like this:

```
120  CREATE a$, DATA,64
```

Since the program was fairly modular, with the file access done in subroutines, the other changes are minimal as well. The idea is to store each item we used before (part number, description, etc.) as a separate data item in the file. Since the part number is always a 4 digit number, we can use an integer to store that

data. Description and location are string quantities, and quantity on hand will fit nicely into a "real" value, since it is a maximum of 5 digits (line 610 checks for that).

The first subroutine to change is the one at line 2000 which writes a record. The new statements look like this:

```
2010 partnum%= VAL(partnum$):quantity= VAL(quantity$)
2020 WRITE#1,rec;partnum%,desc$,location$,quantity
```

There, that was easy. Note that WRITE was substituted for PRINT since this is a DATA file, and instead of packing all the strings together as we did in the old line 2010, we simply converted the string values to the appropriate numeric ones. If we had designed the program to use data files from the beginning, we would probably have handled that in the data entry section of the program.

Next comes the changes to the subroutine which reads a record back. Now things are really very simple. We can replace all the lines between 1820 and 1885 with these:

```
1815 read#1,rec:IF TYP(1) = 5 THEN RETURN
1820 READ#1,rec;partnum%,desc$,location$,quantity
1825 IF partnum%<0 THEN RETURN
1830 partnum$=num$(partnum%)
1840 quantity$=num$(quantity)
```

That's it! Since all the items are stored separately, there is no need to go through the process of splitting them out of the string record. I must confess, however, that I really wanted to discuss the INSTR function, and that previous technique seemed the most logical way to show its features. Oh, well, its always more fun to find an easier way! Two more things are of interest here. Note that we have checked in line 1815 for TYPe 5 which indicates end of file. This takes care of checking for empty records. In line 1825 we introduce a new concept. Previously, when we wanted to delete a record we simply printed a null string over the existing information. There are times when it is useful to only "flag" that a record is deleted, not actually wipe the information out. This allows deleted information be retrieved in the event of mistakes. Periodically another routine can be used which goes through the file and physically deletes the "flagged" records. Here, and below in the actual delete routine, we use making the part number negative to indicate that it is no longer an active record. The "delete" routine now will look like this:

```
1905 partnum%=-partnum%
1907 WRITE#1,rec;partnum%,desc$,location$,quantity
```

That will write the record out with a negative part number, which will flag it as "logically" deleted.

Well, that should just about do it. In addition to giving you several things to try out before next time, the above changes illustrate an important programming "fact of life" (you've been waiting for this article to get juicy, right?). This fact of life is that the more modular your program design, the more painless is the making of inevitable changes. I know that isn't your favorite fact of life, but there is nothing worse than to stare at several thousand lines of BASIC knowing that it has to be completely rewritten.

Next time we'll cover some new but related topics which will require completely rewriting this month's program (just a JOKE!). We will actually talk about different kinds of ways to store and retrieve records on disk which give more flexibility than the simple record number scheme that has been used so far. That should complete the effort to make you a "file expert". In addition, Business Basic has an incredible output formatting capability, and now that you have learned the techniques for storing data, it should be fun to go through some tips on how to make your printouts look like professional reports.

Until then, have fun practicing the "facts of life" (programming facts, of course).

Exploring Business Basic - Part IV

Those of you who have Apple III's have probably received some very good news in the last few weeks. Yes, Virginia, there is a new version of Business Basic with some fantastic new features! But first, a word from our sponsor... Seriously, I would like to conclude presenting the information I promised last time before getting into the new goodies.

As you may remember (and are otherwise encouraged to discover), I concluded the last article by making some points about the use of Data files in Business Basic and modified our simple database program to use the data file format. For at least one more time, I'll list the program as it currently stands and then plunge into this month's enhancements, which cover the breathlessly exciting topics of list management, indexing and sorting. That will about finish us in the data management area, leaving future issues to explore formatting, business arithmetic and matrix arithmetic.

Once more, dear friends, into the breach...

The Program as it Currently Stands

Remember that this program was designed to allow the entry and retrieval of information about parts, such as might be maintained by a distributor or retail store. So far the program allows creation of parts files, and adding, deleting and finding records of specific parts by part number. The program:

```
5   HOME
7   PRINT
10   PRINT"Parts File Create and Modify Program"
20   PRINT:PRINT"Type:"
30   PRINT"     1 to Create a parts file":PRINT
40   PRINT"     2 to Use an existing parts file":PRINT
49   PRINT"     9 to Quit":PRINT
50   PRINT:INPUT"Your selection: ";a$
60   IF a$="" THEN 1000
70   a=ABS(VAL(a$))
80   ON a GOSUB 100,400
90   IF a=9 THEN 1000:ELSE 5
100   PRINT:INPUT"Name of new parts file: ";a$
110   IF a$="" THEN RETURN
120   CREATE a$, DATA,64
130   PRINT"Parts file ";a$;" created."
140   RETURN
400   HOME
```

```
 405   PRINT:INPUT"Name of existing parts file: ";a$
 410   IF a$="" THEN RETURN
 420   OPEN#1,a$
 425   file$=a$
 430   HOME
 435   PRINT:PRINT"Modify Parts File ";CHR$(34);file$;CHR$(34):PRINT
 437   PRINT"Type:"
 440   PRINT"     1 to add to your parts file":PRINT
 445   PRINT"     2 to delete a part from your parts file":PRINT
 450   PRINT"     3 to find a part in your parts file":PRINT
 460   PRINT"     9 to quit the modify mode":PRINT
 465   PRINT:INPUT"Your selection: ";a$
 467   a=ABS(VAL(a$))
 470   IF a=9 OR a$="" THEN RETURN
 475   ON a GOSUB 500,700,800
 480   GOTO 430
 500   HOME
 505   PRINT:INPUT"Part number to add: ";a$
 510   IF a$="" THEN RETURN
 520   a=VAL(a$)
 530   IF a<1 OR a>32767 OR INT(a)<>a THEN 500
 535   rec=a
 540   partnum$=a$
 545   PRINT:INPUT"Description: ";a$
 550   IF LEN(a$)>35 THEN a$=MID$(a$,1,35)
 560   desc$=a$
 570   PRINT:INPUT"Location: ";a$
 580   IF LEN(a$)>15 THEN a$=MID$(a$,1,15)
 590   location$=a$
 600   PRINT:INPUT"Quantity on hand: ";a$
 610   a=0:a=VAL(a$):IF INT(a)<>a OR a>99999 THEN 600
 620   quantity$=a$
 630   PRINT:PRINT"Record is:
";partnum$;"/";desc$;"/";location$;"/";quantity$;
 640   INPUT"  Ok? ";a$
 650   a$=MID$(a$,1,1):IF a$<>"y" AND a$<>"Y" THEN 505
 660   GOSUB 2000
 665   IF errorcode=0 THEN PRINT:PRINT"Record added.":GOSUB 995:GOTO 500
 670   PRINT:INVERSE:PRINT"Record not added, ERROR= ";:NORMAL:PRINT
errorcode:
       GOSUB 995:GOTO 505
 700   HOME
 705   PRINT:INPUT"Part number to Delete: ";a$
 710   IF a$="" THEN RETURN
```

```
 715   a=VAL(a$)
 720   IF a<1 OR a>32767 THEN 700
 725   rec=a
 730   GOSUB 1800
 735   IF errorcode=1 THEN PRINT:PRINT CHR$(7);"No such part
number":GOSUB
995:GOTO 700
 740   PRINT"Delete ";partnum$;"/";desc$;"/";location$;"/";quantity$;"?
";
 745   INPUT"";a$:a$=MID$(a$,1,1)
 750   IF a$<>"y" AND a$<>"Y" THEN PRINT"Not deleted":GOSUB 995:GOTO 700
 755   GOSUB 1900
 760   PRINT:PRINT CHR$(7);CHR$(7);"Record deleted":GOSUB 995:GOTO 700
 800   HOME:PRINT
 805   INPUT"Part number to find: ";a$
 810   IF a$="" THEN RETURN
 815   a=VAL(a$)
 820   IF a<1 OR a>32767 OR INT(a)<>a THEN 800
 825   rec=a
 830   GOSUB 1800
 840   IF errorcode=1 THEN PRINT:PRINT"No such part number":GOSUB
995:GOTO 800
 850   PRINT:PRINT"Part number:      ";partnum$
 855   PRINT:PRINT"Description:      ";desc$
 860   PRINT:PRINT"Location:         ";location$
 865   PRINT:PRINT"Quantity on hand: ";quantity$
 870   PRINT
 890   PRINT:INPUT"Press RETURN to continue: ";a$:GOTO 800
 899   REM
 900   REM delay subroutine
 901   REM
 995   FOR i=1 TO 1000:NEXT i:RETURN
 996   REM
 1000   PRINT:PRINT"End of parts file program."
 1010   CLOSE
 1020   END
 1799   REM
 1800   REM retrieve a record with record number = "rec"
 1801   REM
 1805   errorcode=1
 1810   ON EOF#1 RETURN
 1815   READ#1,rec:IF TYP(1)=5 THEN RETURN
 1820   READ#1,rec;partnum%,desc$,location$,quantity
 1825   IF partnum%<0 THEN RETURN
 1830   partnum$=STR$(partnum%):quantity$=STR$(quantity)
```

```
1890   errorcode=0:RETURN
1899   REM
1900   REM delete a record with record number = "rec"
1901   REM
1905   partnum%=-partnum%
1907   WRITE#1,rec;partnum%,desc$,location$,quantity
1910   RETURN
1999   REM
2000   REM add a record with record number = "rec"
2001   REM
2005   errorcode=0
2010   partnum%=VAL(partnum$):quantity=VAL(quantity$)
2015   ON ERR GOTO 2040
2020   WRITE#1,rec;partnum%,desc$,location$,quantity
2030   OFF ERR:RETURN
2040   errorcode= ERR:OFF ERR:RETURN
```

Impressive, right? In playing around with this program, you may have discovered something very interesting. Retrieving individual records on parts is simple and quick, as long as you remember the part number you want. Try coming back to the program after a few days or weeks (as I have) and attempt to remember the part numbers that were previously entered. The immediate response is that a list of all the active (not deleted) part numbers is needed. The program below will accomplish this task.

```
10   PRINT"Name of Parts file: ";
20   INPUT a$
30   OPEN#1,a$
40   PRINT"Name of list file: ";
50   INPUT a$
60   OPEN#2,a$
70   ON EOF#1 GOTO 1000
75   PRINT"Part number","Description","Location","Quantity":PRINT
80   FOR rec=1 TO 9999
90     READ#1,rec:IF TYP(1)=5 THEN 200
100     READ#1;partnum%,desc$,location$,quantity
110     IF partnum%<0 THEN 200
120     PRINT#2;partnum%,desc$,location$,quantity
200     NEXT rec
1000   PRINT#2:PRINT#2"End of file"
1010   CLOSE
1020   END
```

Notice that this program has been set up to read from any parts file and to output to any file as well. This is similar to some programs from previous articles, and allows to output to go to the screen (by replying ".console") or to a

printer, etc. Additionally, since we have no idea which part number records are actually in the file, a FOR NEXT loop is used to scan all the valid record numbers. Line 90 reads the particular record into memory and checks to see if it contains valid data. Recall that TYP(1)=5 means that there is no data in the record. If data is present, it is read into the variables and the part number is checked. A negative value means that the part number has been deleted. If the data passes all tests, it is printed out.

Running the program reveals several interesting things. Notice the sample printout below:

```
Part number     Description     Location        Quantity

35              shovel          bin 3           2
200             hammer          bin 1           10
300             wrench          bin 5           6
2000            anvil           top shelf       1

End of file
```

Try entering these part numbers yourself and run the sample program. You will notice that the first records print out fairly quickly, but the last one appears only after much whirring of the poor, overworked disk. This is easy to understand, since 1700 records must be searched before the final one is found. Just imagine that I had used 9000 as the last record instead! Clearly there must be a better way to find out what is in the file than searching every record. However, we still want the fast lookup of an individual record which the random record access technique provides.

Here's where all those comments I made earlier in the series about how neat it is to have lots of user memory in Business Basic become important. With the extra memory, we can keep extra data structures around to simplify the task of finding out what data is on the disk and where it is. The term "data structure" is a much revered one in computer science circles, and simply refers to organized ways to maintain data and the information about the data. In this case, we need a structure called a "List". Sounds obvious, right? Lots of things in computer science are needlessly obfuscated (lots of things in English can be too!)

In this case, the list will consist of the part numbers stored in the file. Since the part number is also the record number, our task of retrieving the part number information is simply one of looking up all the record numbers in the list. One other note. The file can contain up to 9999 parts, so it will be convenient to keep track of how many records there are in our list. To do that, the following kind of list will be used:

```
element 1: number of items in the list
element 2: first item record number
```

```
element 3: second item record number
element (number of items + 1):last item record number
```

Since all the record numbers are less than 10000, we can easily use an Integer array to store them and the count. Also convenient is the fact that all arrays in Business Basic begin with element 0, a handy place to store the number of items. The next thing required is a place to store the information permanently so that it can be used by the main program and others (such as the little list program above). This could be done with a separate file on the disk, but it is much safer and more convenient to store the information in the main data file itself. Among other things, it is much easier to keep track of where the information is if it is all physically together. With that in mind, I'll pick record 20000 to park the list. This is clearly out of the way of our regular data, and, because very little extra space is taken up, it doesn't hurt anything.

So much for the philosophy. The following additional program lines will accomplish the task:

First, set up the array for the list:

```
 4   DIM flist%(1000):maxrecord%=1000
```

The variable "maxrecord%" will serve as a check not to exceed 1000 part numbers. Since Business Basic permits very large arrays, this could just as easily been 9999 as long as the DIM statement and the maxrecord% variable agree.

Next, we need to retrieve the list when the file is initially referenced by the program. To allow us to change this easily, a subroutine will be used:

```
 427   GOSUB 2500
 2500   ON EOF#1 GOTO 2570
 2505   READ#1,20000
 2510   IF TYP(1)<>2 THEN flist%(0)=0:WRITE#1;flist%(0):RETURN
 2515   READ#1;flist%(0)
 2520   IF flist%(0)=0 THEN RETURN
 2530   FOR i=1 TO flist%(0)
 2540     READ#1;flist%(i)
 2550     NEXT i
 2560   RETURN
 2570   flist%(0)=0:WRITE#1,20000;flist%(0):RETURN
```

First an ON EOF statement is used in connection with the READ statement in line 2505 to take care of the case where the file is newly created. In that circumstance record 20000 will be past the end of file and statement 2570 will set up the list count in flist%(0) and write that into the file. If record 20000 exists, a check is made to be sure the data is of the correct type (and initialized if not). If everything is ok, the list count is read in and then a FOR NEXT loop

loads the remaining data. Note that this is much more efficient than reading or writing all 1000 values each time.

Next we need to add the "List" option to our menu of things we can do with existing files. Fortunately, the way the program is set up makes that simple to accomplish. The following changes add the new option:

```
452   PRINT"     4 to list the parts in your parts file":PRINT
475   ON a GOSUB 500,700,800,1100
```

The List option requires a new subroutine to read the list array and print the list to the screen:

```
1100   HOME
1110   PRINT:PRINT"List of current parts for parts file: ";file$
1120   PRINT
1130   IF flist%(0)=0 THEN PRINT"No parts on file":GOSUB 995:RETURN
1135   PRINT"Part number","Description","Location","Quantity":PRINT
1140   FOR i=1 TO flist%(0)
1150     rec=flist%(i)
1160     GOSUB 1800
1170     IF errorcode=0 THEN PRINT partnum$,desc$,location$,quantity$
1180     NEXT i
1190   PRINT:INPUT"End of list, press RETURN to continue: ";a$
1200   RETURN
```

After first checking to see if the list was empty, the heading is printed and the list array is scanned. We can use the subroutine at 1800 to retrieve the record and set up the variables. That subroutine also checks for deleted records and line 1170 uses the errorcode variable to check for that. Note that we could have opened a secondary file here to redirect the list to another device if desired.

The next changes just clean up some sloppy programming from before. See there? There is no such thing as a perfect program (or a perfect programmer). These changes just recognize the fact that our part numbers were supposed to be four digit numbers, yet we allowed any part number up to 32767. That was fine before we decided to put the part number list at record 20000. The changes are as follows:

```
530   IF a<1 OR a>9999 OR INT(a)<>a THEN 500
720   IF a<1 OR a>9999 THEN 700
820   IF a<1 OR a>9999 OR INT(a)<>a THEN 800
```

The next change is to add to the list each time a record is added to the file.

This involves updating the list count and storing the new part number in the next available list position. Since adds are done in the subroutine at line 2000, the changes are simple:

```
2006   IF flist%(0)=maxrecord% THEN errorcode=-1:PRINT"Parts file
```

```
full":RETURN
 2025   flist%(0)=flist%(0)+1:flist%(flist%(0))=rec
```

First, line 2006 checks to be sure that the list count will not be exceeded by adding this record. If everything is ok, line 2025 updates the list count and uses it as the index to store the new part number (record number).

The last task is to write out the updated list as a part of ending the program. This must also be done when switching to a new file. The changes are as follows:

```
 470   IF a=9 or a$="" THEN GOSUB 2600:RETURN
 1005   GOSUB 2600
```

The subroutine at 2600 does just the reverse of the one at 2500, that is, writes the list back into the file starting at record 20000:

```
 2600   IF file$="" THEN RETURN:ELSE:WRITE#1,20000;flist%(0)
 2610   IF flist%(0)=0 THEN RETURN
 2615   FOR i=1 TO flist%(0)
 2620     WRITE#1;flist%(i)
 2625     NEXT i
 2630   RETURN
```

Notice that we use the fact that file$ is assigned the name of the file after opening to determine if the modify section of the program was used. If the string is empty ("null") then there is no open file to which to write.

All that above seems like a tremendous number of program changes, I know, but the results are well worth it. You can now find out what's in the file at any time, and the listing speed is essentially independent of the way the data is arranged in the file. Furthermore, this permits us to do some really interesting things later.

The type of data structure used here is commonly referred to as a "variable length list". Here "variable" is used in the sense of "changeable". This is an extremely useful and widely used structure. One example in front of you at the moment is the Business Basic string variable. See your Basic manual for details of how the length of a string is stored.

INDEXING AND SORTING

Now that we've made all these fun changes, try running the program on a new file. Try adding the following part numbers in this order: 5,35,200,100,50. Now when you use the list option, you will notice that the part numbers appear in the order in which they were entered. The previous example of a separate list program always listed the parts in part number order, since it scanned the file sequentially from the beginning. Ordering of lists according to the sequence in which they were entered into the file is called "chronological" order. Ordering the list in any other way is generally referred to as a "sorted" order.

Clearly, if the array "flist%" was arranged in numeric order, we could use the subroutine at 1100 to list the contents of the file out in that order. That's because the values in flist% are used as "indexes" into the larger file itself. It is the value assigned to the variable "rec" in line 1150 that determines which record is read and listed. Unfortunately, sorting the information in flist% would destroy the chronological order, and that might be a useful way to list the data as well. This implies that we should create some additional arrays to hold sorted versions of the flist% array. These arrays are sometimes called "sorted indexes". In fact, it may occur to you that several of these sorted indexes could be stored simultaneously in the file. Similar kinds of "multi-key indexing" are used in sophisticated database management systems.

Wow! That's a lot of definitions of esoteric computer topics. In fact, there is enough implied in the paragraph above to be the meat for several articles. We'll look at a simple example and then I suggest you slide over to your local library for a book on database techniques for the real details.

First, let's change the list routine to provide some sort options:

```
 1102   PRINT"Type:"
 1103   PRINT"     1 for chronological order"
 1104   PRINT"     2 for part number order":PRINT
 1105   INPUT"Your selection: ";a$
 1106   sortorder=VAL(a$):IF sortorder<>1 AND sortorder<>2 THEN GOTO
1100
 1107   GOSUB 1300
 1140   FOR i=1 TO slist%(0)
 1150     rec=slist%(i)
```

The changes from 1102 to 1107 set up the choice and GOSUB to 1300 to perform the actual sort. Lines 1140 and 1150 change the list index to a new array "slist%" which has the same structure as flist%. This allows changing the order without changing the actual contents of flist%. This also means a change to line 4:

```
 4   DIM flist%(1000),slist%(1000):maxrecord%=1000
```

Isn't having all that memory nice?

Next, lets cook up a subroutine which will sort the flist% array and create a slist% array with the contents in ascending order:

```
 1300   IF flist%(0)=0 THEN RETURN
 1305   slist%(0)=flist%(0)
 1310   FOR i=1 TO flist%(0)
 1315     slist%(i)=flist%(i)
 1320     NEXT i
 1325   IF sortorder=1 THEN RETURN
 1330   length%=slist%(0)
```

```
 1332   IF length%=1 THEN RETURN
 1335   FOR pass=1 TO length%:madeaswap%=0
 1340     FOR position=1 TO length%-pass
 1345       IF slist%(position)>slist%(position+1) THEN SWAP slist%
(position),slist%(position+1):madeaswap%=1
 1350       NEXT position
 1355     IF madeaswap%=0 THEN RETURN
 1360     NEXT pass
 1365   RETURN
```

Several things are of note here. First, if there is anything in the flist% array, it is copied to slist%. If chronological order is desired, we're finished. If not, the contents of slist%, but not the list count, slist%(0), must be sorted in order. For simplicity, we use a version of the classic "bubble" sort, with a new wrinkle. Business Basic has a new statement named SWAP which comes in very handy in sorting situations, among others. It will exchange the values of any two variables of the same type. This includes, as this example points out, elements of arrays. Normally this exchange is handled by assigning one variable to a temporary variable, as in the following example:

```
1345   IF slist%(position)>slist%(position+1) THEN temp%=slist%
(position):slist%(position)=slist%(position+1):slist%(position+1)=temp%:
madeaswap%=1
```

In addition to being ugly, this version performs significantly slower than the version using SWAP, since SWAP is done internally by Basic in assembly language.

Try putting this routine into your program. For small lists it will perform very well. For larger lists, there are far better sort techniques. Later in this series I will try to do an article on different sort techniques. Most microcomputer references on sorting tend to try to minimize memory utilization, which usually hurts performance. Since you lucky Apple III owners have fewer problems in that area, the classic techniques have to be looked at differently.

It might also occur to you that it is possible to sort on items other than the part number. A good experiment for you might be to change the sort subroutine so that the slist% array was used to read in records to build a string array from the values of "description$". When you sort the string array, you could swap the slist% elements in correspondence to the way you swap the string array elements. Then listing from slist% would produce a list in description order. This is referred to as a "pointer" sort.

Another interesting change would be to use record 0 of the data file as a place to store the record number where flist% begins. Right now that is "hard coded" at record 20000, but for a lot of reasons, it might need to be changed later. Writing it into the file and reading it back at open time would make the program much more flexible. Also, if you decided later to keep multiple indexes for

different elements, you could store all their "addresses" there (or maybe just the address to the addresses!). Another thing hardcoded into this program is the record format, including not only the number of elements, but their names, type and range of values allowed. Real database programs maintain this information in the file as well, permitting the user to define many different databases with the same program. You might think about how our program would be modified to do that as well.

The paragraph above contains enough challenges to last you as long as you want. Just remember that Business Basic has enough power and capability to allow you to be as sophisticated as you wish in managing file data. Good luck!

THE NEW GOODIES

Version 1.1 of Business Basic is now released and it is neat! Obviously, it clears up some nasty little problems from the first version, and the manual is now a completely revised (and two volume) reference guide that you will really enjoy. But that's only the beginning. Several new capabilities have been added in response to user requests and some pretty good thinking on the part of the Apple engineering staff. They are summarized below, but I suggest that you slide over to your local dealer to get the real scoop. The P.S. is pretty good too. Its free to all current purchasers of Basic, no matter how long ago you bought your old version!

New language additions

There are two new reserved words, INDENT and OUTREC. INDENT sets the level of indentation for the FOR NEXT loops (default is 3) and OUTREC sets the record length Basic uses to format listings. Ever have a long line in a Basic program which got overprinted on your 80 column printer? OUTREC is initially set to 80, but can have any value to 255. Zero causes listings to work as in the old version. The neat feature is that when the printed output reaches the OUTREC value, Basic automatically inserts a carriage return and spaces over to line the next part up with the indentation level of the previous line. No more screwing up those pretty indented listings with long lines! This works with any output file you specify.

An enhancement has been made to the GET statement as well. You are now allowed to use GET# to get a single character from any file. This includes disk Data files, Text files, and character devices. I can't begin to tell you all the possibilities this presents, but it should keep you busy for a while.

NEW INVOKABLE MODULES

As we've discussed before, the design of the Basic Invoke mechanism allows the language to be extended almost infinitely. Since the Invokable routines are accessed by name, and available from Immediate as well as Deferred execution modes, its really like adding commands to the language. With all the memory available in the Apple III, you can keep lots of these routines around, or if you need the space for data, you can invoke just the ones you need at a particular time. The new release of Business Basic contains some really powerful Invokables. Hang on to .console, here they come!

For the development programmer, the most significant module is probably RENUMBER/MERGE. There's too much available in this one to go into detail but for those of you who have been frustrated by wanting to add that "one extra line" into a program and having no place to put it, take heart. RENUMBER will renumber your program in memory and save it on disk automatically, or renumber a program stored on the disk and place it in memory for you. In addition, it supports merging of programs on disk with programs in memory. This means you can save important subroutines and have them automatically added to the program you are currently working on. Because it is an Invokable module, it won't take up any memory unless you want it (obviously there is little need, and less desire, to renumber a program while it is running).

The next biggie is the REQUEST module. Remember all those wonderful things I keep saying about SOS? REQUEST allows a Basic program to make calls to the Operating System directly. You can read or write up to 64K bytes in one statement to any file on the system (including text and data files, .console, etc.). Numeric arrays can be stored about 20 times faster than with regular FOR NEXT loops! In addition, REQUEST allows the Basic programmer to directly get device status, and use the SOS SETCONTROL mechanism. More details on this super-powerful module are in the documentation.

The last goodie is an invokable which allows you to DOWNLOAD character sets directly to the RAM-based character generator. After setting up the character definition in an array, one Perform statement passes it to the Operating System as the new character set. The Business Basic disk contains several sample character sets, and you can have fun inventing your own. This also allows you to create animation, by properly defining special characters, ala the DOS Toolkit ANIMATRIX program. I'm sure some clever programmers will design a really nice program to use this invokable for character set design.

Closing thoughts

Whew! Glancing up at the prompt line of my Applewriter III display, I see that I'm up to 26590 characters in this article! Your tired eyes and my tired fingers

both need a rest. Next time we'll have a mixed bag of things to enjoy, including some comments on the powerful formatting capabilities of Business Basic, and I'll reveal a secret that I hope you all get in on. That secret is the answer to the question "how many bytes of memory are available for programs in a 256K Apple III?". Until then, have a happy holiday season!

P.S. As you probably noticed in the paragraph above, there's more than Basic that's changed about the new Apple III!

Exploring Business Basic - Part V

Last time I dropped several broad hints about new software and hardware happenings on the Apple III. Hopefully by now you have had a chance to go down to your dealer and check some of these things out. As you might have guessed, I've been doing these articles on Applewriter III, which I really enjoy. I could go on describing the new software for the rest of this column, but since this is supposed to be about Basic, I'll temporarily restrain my enthusiasm. One thing before we start, though. Last time I promised the answer to "How many bytes of memory are available in a 256K Apple III?" Well, as you know, the 256K Apple III has been announced and is beginning to be available. The answer can now be revealed: 191,484 bytes! That's over three times the workspace available in any other personal computer Basic. (Aren't you glad you've got an Apple III? Don't you wish everybody did?) We were discussing some sorting techniques for our database last time which can make good use of that space. This time we'll explore a mixed bag of items, and defer our discussion of the Print Using capabilities of Business Basic until next time.

Our Mixed Bag

The first bagged item for this month is the mailbag. Several questions have come my way since this series started in September, but the most interesting ones concern programming style and philosophy. The most intriguing concerned why I always use lower case variable names in my programs, especially since the Basic keywords (PRINT, etc.) all seem to be in caps. Actually, it would be easy to say that I lack the strength or will to operate the alpha lock key, but the real reason has to do with the way Basic itself works. You are probably aware that Business Basic defers its "syntax checking" (looking for errors) until you actually run the program. Basic does perform some tasks as each statement is entered, however, a process generally referred to as "tokenizing". Simply, this means that Basic scans each statement and converts each Keyword, sometimes called Reserved Words, into a special internal one byte code called a Token. English majors will appreciate the appropriateness of that term. Not only does this code save space, but it simplifies error checking and execution of the program. Almost all Basic interpreters use this technique. One of the consequences of this method is that the program statements cannot be listed out without the Basic LIST command converting these "tokens" back to their English equivalents. In converting the tokens, Basic always prints out the Upper case version of the keywords. I type in all Basic statements in lower case, both variables and keywords, so that when I list out the program, I can see what Basic interpreted as keywords. If I misspell PRINT, Basic will not recognize it as a keyword, and the fact that it remains lower case makes such errors easy to spot in a listing. In addition, Business

Basic requires spaces between keywords and variable names, to allow variables to contain keywords themselves. Ever try to use a variable like ORANGE in Applesoft, only to discover that OR is a reserved word and therefore your variable must be renamed to something like RNGE? Typing in lower case will allow you to spot those times that you forgot to space and ended up with "fori=1 TO 10" instead of "FOR i=1 TO 10". The first case will produce an error, since Basic will assume you are trying to assign the value of 1 to the variable "fori" and for some reason put the phrase "TO 10" onto the end of the statement. Some examples will clarify:

```
  Typing:  10 prunt x*53     will result in:  10 pruntx*53

 whereas:  10 print x*53     will result in:  10 PRINT x*53

  Typing:  10 on xgoto 20,40,50   will result in:  10 ON xgoto20,40,50

 whereas:  10 on x goto 20,40,50  will result in:  10 ON x GOTO 20,40,50
```

See how much easier it is to catch the error visually? Like every rule, there are exceptions. Any variable which starts with the letters "FN" will be assumed to be a function name. Again, typing all lower case will help spot the problem:

```
Typing:  10 xval=aval*fnumber    will result in: 10 xval=aval* FNumber
```

and you will immediately know that something is wrong (assuming that you really wanted to use "fnumber" as a variable name).

There is another little quirk in Basic that this technique helped me spot. As you may know, we have used the ON EOF# statement quite a bit to take action if a program tries to read past the end of file. According to the manual, the part following EOF#n can be any executable statement. So far we have generally used GOTO or GOSUB statements to take action. Consider the following:

```
 Typing:   10 on eof#1 goto 20   will result in:   10 ON EOF#1 GOTO 20
```

as you would expect. But:

```
    Typing:   10 on eof#1 xval=20   will result in:   10 ON EOF#1xval=20
```

For some reason Basic treats the whole thing as one variable. The solution involves dredging up a bit of Basic folklore. Remember in your first class in Basic when they told you that all assignment statements started with the keyword LET? Most Basic dialects have long since allow the LET keyword to be optional, and most people have quit using it altogether. An example of the use of LET is:

```
  10 LET x=45    which is usually written simply:   10 x=45
```

If there is any ambiguity in the way a statement can be interpreted, LET can be used to clear it up. Our new version of the EOF statement:

```
typing:   10 on eof#1 let xval=20  will result in:  10 ON EOF#1 LET
xval=20
```

and everything will work fine. Note that the fact that Basic failed to upshift the reserved word EOF in the example above was a major clue toward understanding the problem. The technique of entering everything in lower case has saved me countless hours of debugging my errors. I recommend it strongly to you.

Bag Item Number Two

In my list last month of new goodies in Business Basic 1.1, I completely overlooked one which seems minor but has important consequences. The change is an extension to the standard GET statement. Normally, as in Applesoft and some other Basics, GET allows reading the keyboard one character at a time, including all special control characters and delimiters. This means you can bypass Control-C and Carriage Return, read commas, etc. Business Basic 1.1 extends GET to allow GET#n. This means that you can read any SOS file one character at a time, without respect to what kind of file it is. This can be very handy to read all characters from the communications port (via the .RS232 driver) or read other character streams from special devices. One of its most interesting uses, however, is the fact that it can be used on disk files as well. Remember that one file is just like another in the SOS environment, so if we open a text file on disk, GET# will allow us to read one character at a time from it. This means that there is now an easy way to read text files which contain more than 255 characters without a RETURN character. Normally this would cause a string overflow error if attempted with the Basic INPUT statement. Even more interesting is the fact that we can also open and read from the Basic Data file! Remember that I described the Data file as having special "tags", called "Type bytes", to allow Basic to determine what data type was stored next in the file. Remember also that numeric data is stored in a Data file in its binary form. GET# allows reading this binary information, one byte at a time. One example is worth a thousand explanations:

```
5   INPUT"File to dump: ";a$
10   IF a$="" THEN 100
15   OPEN#1,a$
20   ON EOF#1 GOTO 100
25   cr$=CHR$(13)
30   GET#1;a$
40   IF a$=cr$ THEN PRINT
50   PRINT a$;
70   GOTO 30
100   CLOSE
110   END
```

This simple example will dump any text file to the screen, no matter how long the intervals between carriage returns. A good example of a text file with arbitrarily long strings is the file I'm creating now, using Applewriter III. RETURN characters are inserted only at the end of paragraphs which, if you notice my style, tend to run on indefinitely.

Note that in this program I look for RETURN characters by loading the variable "cr$" with a RETURN (decimal 13) and then testing for it before printing. If you want to reconstruct strings from the file, you could use a string variable to accumulate characters, stopping when a RETURN is encountered. A test would need to be made to avoid overflowing the 255 character limit.

This program has one serious deficiency, however. Printing arbitrary characters from a file (especially a Data file) can have weird consequences when the output device is the console, as it is in the example program. The console uses lots of different control sequences to perform functions, including setting windows and changing from black and white to color text modes. Also, a byte can contain 256 different characters, and the ASCII character set only defines 128. Clearly we need a safe and consistent way to display any byte readable from a file. So, like most small, simple programs, this last one is about to get complex:

```
5   INPUT"File to dump: ";a$
10   IF a$="" THEN 95
15   OPEN#1,a$
20   INPUT"File for output: ";a$
25   OPEN#2,a$
30   ON EOF#1 LET eof.occurred=1:GOTO 80
35   bytes=0:eof.occurred=0
40   line$=""
45   PRINT#2;HEX$(bytes);"-";HEX$(bytes+31);" ";
50   FOR i=1 TO 32
55     GET#1;a$
57     val=ASC(a$):IF val>127 THEN val=val-128
60     IF val<32 THEN line$=line$+" .":ELSE:line$=line$+" "+CHR$(val)
65     outhex$=HEX$(ASC(a$))
70     PRINT#2;MID$(outhex$,3,2);
75     NEXT i
80   PRINT#2:PRINT#2;"          ";line$
85   bytes=bytes+32
90   IF eof.occurred=0 THEN 40
95   CLOSE
120   END
```

Scanning down the program, note that in addition to opening the file to be dumped, we now open a second file to which the output is written. This gives more flexibility, and still allows using ".console" to see the output on the screen. Line 30 sets up our end-of-file condition, using the LET statement to get around

the problem we described earlier, and shows one other handy thing of note. Notice that we can imbed periods in variable names to improve readability. It's easy to see that "eof.occurred" is easier to interpret than "eofoccurred", and this is especially true for more complex variable names (remember that Business Basic permits 64 character names!).

Lines 35 and 40 initialize variables. We will be using the "line$" string to accumulate the characters read from the file for later printing. After each line of print we will re-initialize the string. Since we will be printing 32 characters at a time from the file, line 45 uses the HEX$ function to set up the labels for each line. A note about "HEX" is appropriate here. HEX stands for hexadecimal, or base 16 arithmetic. It is used extensively in computers for its convenience, since any hex digit can be represented by 4 binary bits, and a byte can be exactly represented by 2 hex digits. This convenience makes it preferred over decimal and octal notation, and of course it is much more compact than binary. What usually throws people is that to properly represent with a single digit all values between 0 and 15, hex uses the digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F respectively. "F" thus is equivalent to 15, and "1F" is 31 (the 1 is in the "sixteens" place). There will be no attempt to explain hex further. If you are not familiar with the notation, any beginning text on computers usually covers the subject thoroughly, and readers of Rodger Wagner's column in this journal have been inundated with help on "hex". Suffice it to say that the HEX$ function will convert any reasonable numeric quantity into a four byte string of hex digits.

Getting back to our program, the loop from line 50 to line 75 is the main one where we dump 32 bytes at a time in hex format, while providing character representations for those within the displayable range (hex 20 to 7F, decimal 32 to 127). The back of your Basic manual contains an ASCII code chart, which will be helpful in following along with the decoding. Line 57 in the program sets the variable "val" to the ASCII value of the byte just read, and then an IF statement checks to see if it is in the 128 to 255 range. If so, 128 is subtracted to bring the value within the normal ASCII range. Line 60 checks to see if the resulting character is a "control" character, and if so, it is represented as a period in "line$", signifying that it is unprintable. Otherwise, the character representation is stored. Note that the characters are right justified in each two byte cell, because they will be printed below the hex values. Next, the hex value of the original character is assigned to "outhex$" in line 65, and printed to the output file in line 70. Note that we want only the rightmost two hex digits, so the MID$ function is used. After the loop prints out the 32 values, lines 80-90 print the ASCII equivalents stored in "line$", bump the byte count, check for EOF condition, and repeat the sequence.

The easiest way to check this little jewel is to run it against the file for this article. If we did that, the output would look something like this:

```
0000-001F
2E636A0D5420482045202054204820492052204420420422041205320492043 0D
          . c j . T   H   E     T   H   I   R   D     B   A   S   I   C
.
0020-003F
0D6279205461796C6F7220506F686C6D616E0D0D0D2E6C6A0D4578706C6F7269
          . b y   T a y l o r   P o h l m a n . . . . l j . E x p l o r
i
0040-005F
6E6720427573696E6573732042617369632D20506172742066697665 0D0D4C
          n g   B u s i n e s s   B a s i c   -   P a r t   f i v e . .
L
0060-007F
6173742074696D6520492064726F707065642073657665726 72616C2062726F6164
          a s t   t i m e   I   d r o p p e d   s e v e r a l   b r o a
d
0080-009F
2068696E74732061626F7574206E657720736F66747761726520616E64206861
            h i n t s   a b o u t   n e w   s o f t w a r e   a n d   h
a
00A0-00BF
726477617265206861707065 6E696E6773206F6E2074686520417070 6C65202F
          r d w a r e   h a p p e n i n g s   o n   t h e   A p p l e
/
00C0-00DF
2F2F2E2020486F706566756C6C79206279206E6F7720796F7520686176652068
          / / .     H o p e f u l l y   b y   n o w   y o u   h a v e
h
00E0-00FF
6164206120636861 6E636520746F20676F20646F776E20746F20796F75722064
          a d   a   c h a n c e   t o   g o   d o w n   t o   y o u r
d
```

Messy, huh! Let's look at this more closely and see if it makes sense. First, the first line tells us that we are looking at bytes 00 through 1F (0 to 31 decimal), and the top line is the hex representation of the characters, two digits per character. The first character in the file is "2E" hex, which happens to be a period. Notice that that is the character printed below on the next line. The next two characters in the file are "63" and "6A" which correspond to the ASCII characters "c" and "j". This is understandable, since Applewriter III uses the print format command ".cj" for center-justify, which is what I wanted done with the title. The next character is "0D" which translates to decimal 13, or a RETURN character. Note that a period is substituted for this character on the print line, since RETURN is in the "control" character range. And so on, and so on. Practice on a few text files of your own and get a feel for reading the notation.

Where this really gets interesting is in reading files whose exact format is normally pretty obscure. Data files are an excellent example, since, although the READ# statement can get data out, things like the "type" bytes, and string length bytes are normally inaccessible. To see how our dump program would work on a Data file, we need a way to generate an interesting file at which to look. The following program is simple, and will do the trick. Later on in a future article when we get serious about sorting techniques, we'll need such a program, so I'll introduce it now:

```
 5   OPEN#1,"junkfile",30
 6   INPUT"Number of records to create: ";n
 10   FOR i=1 TO n
 12     i%=RND(1)*10000
 13     WRITE#1,i;i%:PRINT i%,
 15     a$=""
 20     FOR j=1 TO 5
 30       a$=a$+CHR$(65+INT(6*RND(1)))
 35       NEXT j
 41     FOR k=1 TO 4
 42       a$=a$+CHR$(48+INT(10*RND(1)))
 43       NEXT k
 45     WRITE#1;a$:PRINT a$,
 48     val=RND(1)*1E10:WRITE#1;val:PRINT val,
 49     i&=CONV&(RND(1)*1E15)
 50     WRITE#1;i&:PRINT i&
 55     NEXT i
 60   CLOSE
 70   END
```

This program will create a random access data file of arbitrary length containing an integer, a string, a real and a long integer in each record. The interesting things of note are the two small loops which build the string value. They are set up in such a way to insure that the first five characters are upper case alpha, and the next four are decimal digits. As I said, this routine will come in handy later in sorting exercises. For now, type this in and run it to create a small file, say 5 records. Although each run will differ, the output should look something like this:

```
2092            CEEBE4542       7.72055E+09     930904428626944
7107            CDCAD1031       6.87212E+09     971614244086784
9206            DDADE8239       6.94853E+08     839965717072896
3038            ADBAC4450       6.09472E+09     397952126404096
3814            AABED9057       2.27867E+09     768212125296640
```

Now for the fun. When you run your dump program against the file that this program creates, the output should look something like this:

```
0000-001F
00000000000000000000000000000000000000000000000000000001208
          . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
.
0020-003F
2C21094345454245343534214A16617221800034EA713C9C4000000121BC321
          , ! . C E E B E 4 5 4 2 . ! f . " . . . N ' . I D . . . . . C
!
0040-005F
0943444341443130333114A14CCE1D18000373AD91E0A40000001223F6210944
          . C D C A D 1 0 3 1 . ! L N . . . . s - . ` $ . . . . # v ! .
D
0060-007F
444144453832333914 9E25AA7B180002FBF1C2308C000000120BDE2109414442
          D A D E 8 2 3 9 . . % * { . . . { q B 0 . . . . . . ^ ! . A D
B
0080-009F
41433434353014A135A30818000169EF7321C6000000120EE621094141424544
          A C 4 4 5 0 . ! 5 # . . . . i o s ! F . . . . . f ! . A A B E
D
00A0-00BF 3930353714A007D1D0180002BAAF52D728000000
          9 0 5 7 .   . Q P . . . : / R W ( . . .
```

Well, nobody said computer science was for the faint of heart! By the way, the term generally used for this type of listing of file contents is "formatted dump". "Formatted" because we have organized the information in the printout, and "dump" because it is a non-selective output of the exact contents of the file.

Now the fun begins. The first thing to notice is that almost the entire first line is composed of zeros. Remember that our dump program starts at the beginning of the file, but the program we used to create this file began at record number one. Therefore, since the record size was 30 bytes, we would expect to find an empty record of 30 bytes at the beginning. That is exactly what the dump shows. This means that the hex "12" in byte 31 of the file must be at the beginning of record one. Now something that was mentioned earlier about "type" bytes in Data files becomes important. Remember that the general format of a Data file is the following:

```
!==============!========================!
! Type byte    !  Data bytes (2,4 or 8)  !
!==============!========================!
```

for numeric values (integer, real and long integer), and the following:

```
!==============!==============!========================!
! Type byte    ! Length byte   !  Data bytes (0 to 255)  !
!==============!==============!========================!
```

for strings. With this information, we should be able to decode the information in this dump.

Since the first value in the record was an integer, the hex code "12" must be the "type byte" for integer data. Following our format, that means the next 2 bytes (hex codes "08" and "2C") must be the binary integer value. Evaluating the hex value "082C" yields decimal value 2092, which is exactly what our printout led us to expect.

The next value in the file was a string, which contained "CEEBE4542". Referring again to our format for strings in Data files, we would expect the next file byte to be the "type" byte. That's the hex code "21". Next is the length byte, which, since the string is 9 characters long, should be equal to 9. That's hex code "09", one of those lucky hex numbers which is the same as its decimal equivalent. After that our format line shows that indeed, the string value is "CEEBE4542".

The next value in the record was a "real". Since the next byte after the string should be the "type" byte for reals, we can conclude that the hex "14" found in position "2C" (44 decimal) is the floating point "type" byte. Floating point numbers are stored in a 32 bit internal format in Business Basic, so we would expect that the next four bytes would contain the binary value. Proving that this value (hex "A1661722") is equal to 7.72055E+09 is considerably more complex, and will be left to the numerically inclined reader. That phrase "left to the numerically inclined reader" is this author's equivalent to the famous line found in all math texts "it can easily be shown that..." and is just as big a cop-out.

The last value in the record is a long integer, and the "type" byte in position "31" (decimal 49) has the value of hex "18". Long integers are stored as eight byte quantities, therefore the next 16 hex digits should represent the number. Since that hex value is "00034EA713C9C400", it follows that coverting this value should yield the decimal value originally printed out: 930904428626944.

As a little added bonus in this article, let me offer a program which will demonstrate the truth in the statement above. This program will convert any reasonable hex value into decimal and print it out rather quickly, using the long integer data type and Business Basic's conversion functions. Forthwith, it is:

```
5   sixteen&=16
10   INPUT"hex value: ";a$
15   IF a$="" THEN 100
20   cum&=0
```

```
25   mult&=1
30   FOR i=LEN(a$) TO 1 STEP-1
35     val&=CONV&(TEN(MID$(a$,i,1)))
40     digit&=mult&*val&
45     cum&=cum&+digit&
50     mult&=mult&*sixteen&
55     NEXT i
60   PRINT cum&
65   GOTO 10
100   END
```

The program simply "brute force"s the problem, one digit at a time, but since the long integer arithmetic is very fast, performance of the program is quite reasonable. One note, this program knows nothing about sign bits, so it will fail in converting negative integers expressed as hex constants. A fix for this would be to optionally check for the high-order bit and negate the final result, but the program then loses its general nature. Anyway, it's free.

Well, that got us completely off track. Going back for a second to the formatted dump, we are now at position "3A" hex (58 decimal), which is really position 28 decimal in this record. The remaining two bytes of the record (remember that we declared the record to be 30 bytes long) should be empty, and sure enough, show up here as zeros. This gets us to position "3C", the beginning of the next record, and there is the integer "type" byte "12" signaling that we can start the whole process again. I leave that to you if you want to try your hand at decoding. I can summarize some of what we have learned in the following table:

Data type name	TYP() function value	Internal file code	
		hex	decimal
Integer	2	12	18
Real	1	14	20
Long Integer	3	18	24
String	4	21	33

Don't forget that the GET# statement can be used in lots of other interesting ways and that its primary function is to effectively process console input without those characters being first processed by Basic. I just thought the examples above would give us a chance to explore several interesting topics at once.

Final Thoughts (Bottom of the Bag)

I had fully intended to explore one more topic which had previously generated questions, but this tome is now growing overlong. The topic I had in mind was the use of the REQUEST invokable module. Those of you who are writing programs which do lots of reading and writing of numeric arrays to disk should tune in next time when we show how to get at least twenty times the performance improvement over using FOR NEXT loops to accomplish the same task. That, combined with the huge memory space available for arrays, provides some significant capability to the person interested in data analysis and sophisticated file indexing. I also promise to get to my thesis on PRINT USING, especially since Business Basic allows some tricks not available in most other Basics. One of these days we'll get to graphics as well, and discuss how to use BGRAF and DOWNLOAD to create some really interesting stuff.

Until then, just one last note. I looked back over this article and decided that the word "hex" was mentioned so many times that we have left the era of "Voodoo Economics" (an unpopular phrase in Washington these days) and entered a new era of "Voodoo Basic". Oh well, maybe if I wore garlic while typing...
files, .console, etc.). Numeric arrays can be stored about 20 times faster than with regular FOR NEXT loops! In addition, REQUEST allows the Basic programmer to directly get device status, and use the SOS SETCONTROL mechanism. More details on this super-powerful module are in the documentation.

Exploring Business Basic, Part VI

Last episode I covered a mixed bag of topics, and ended with a promise to cover some parts of the new REQUEST invokable module and techniques on using PRINT USING. Fear not, all that and more is covered below, including some tips on Long Integer "decimal" arithmetic. But first, a few digressions based on comments some of you made on previous articles.

Digression Number One

I made the comment in one of the first articles that random record files were limited to 32,767 records, the maximum positive integer value. In fact, there is no particular limit in SOS on which this Basic limit is based. Basic even allows a real number to be used as a record number, but because Basic uses an integer type internally to keep track of the record number, the value still cannot exceed 32,767. The actual position in the file is determined by multiplying this record number by the record size assigned when the file was originally created (default is 512 bytes). My understanding is that this 32,767 limit on record numbers is not the case in Pascal. Now that the Profile hard disk is available, some thought is being given to removing this restriction. Let me know if it's been a problem for you.

Digression Number Two

As I demonstrated last time, the GET# statement in Basic can be used to read the exact contents of most files on the Apple III, one byte at a time. We even created a special "formatted dump" program to investigate the contents of Basic Data files. Some types of disk files cannot be opened by Basic, however. Most notably, these include Pascal Code and Data files. If you have a need to examine the contents of those files from Basic, they can be accessed by using the Pascal Filer to change the file type to "Ascii", which Basic knows as the "Text" file. You Pascal programmers will enjoy Basic, once you try it!

There is another file type which is very interesting to examine, and Basic will allow you to open it directly. Those are the "Catalog" or "Subdirectory" files which you create from Basic or the Utilities program. The subdirectory capability of SOS is one of its most powerful features. If you aren't using subdirectories to group your files and programs logically, I suggest you read the relevant sections of the Basic manual and the Apple III Owners Guide. One problem with files in Basic, however, is that it is difficult to discover from a running program whether or not a given file or program already exists. There are some ways using ON ERR to work up a solution to this problem, but nothing very tidy. However, being able to open and read a directory or subdirectory

allows us to check on everything before OPENing a file or CHAINing to another program.

Those of you who read last month's article know about our handy-dandy file dump program using GET#. Let's pick a typical subdirectory named "MYSUB" containing the files "MYPROGRAM" and "DIRECTORYDUMP". Using the formatted dump program from last time, the file contents look something like this:

```
0000-001F
00000000E54D5953554200000000000000000076000000000000009DA3060F
          . . . . e M Y S U B . . . . . . . . . . v . . . . . . . . # .
.
0020-003F
000000270D020009000227194D5950524F4752414D0000000000000982000100
          . . . ' . . . . . . ' . M Y P R O G R A M . . . . . . . . . .
.
0040-005F
A301009DA3070F0000E300029DA3070F81002D4449524543544F525944554D50
          # . . . # . . . . c . . . # . . . . - D I R E C T O R Y D U M
P
0060-007F
000004840006007609009DA3090F0000E300029DA3090F810000000000000000
          . . . . . . . v . . . # . . . . c . . . # . . . . . . . . . .
.
```

For those of you who did not read last month's column, this may look bizzare, but it's really easy. Remember that this is a byte-by-byte image of the file. The numbers to the left (like 0000-001F) are the byte numbers in hexadecimal of that particular row. Each row contains 32 bytes. The top row in each pair is the actual hex contents of the file, and the next row is the ascii equivalent characters. If the byte is a non-printing character, it is represented by a ".". This is all fine, but you will immediately protest that other than being able to spot the subdirectory name and the file names, the printout is a big mystery. Business Basic to the rescue! It turns out that Basic is knowledgeable of the contents of directory files, so that when you open a directory or subdirectory file, Basic will automatically format the contents for you, just as it does in the CATalog command. The following simple program will illustrate, on the same subdirectory we just looked at:

```
1   INPUT"Directory to dump: ";a$
10   OPEN#1,a$
15   ON EOF#1 GOTO 60
20   INPUT#1;a$
30   PRINT LEN(a$)":"a$
50   GOTO 20
60   CLOSE
```

```
 70   END
```

The only thing unusual here is that we arranged to print the length of each string that is read, to check for any special formatting. The output looks like this:

```
68: MYSUB           (12/29/81) V0
68:
68:  TYPE   BLKS  NAME            MODIFIED TIME  CREATED  TIME   EOF
65:  BASIC  00001 MYPROGRAM       12/29/81 15:07 12/29/81 15:07  419
66:  TEXT   00006 DIRECTORYDUMP   12/29/81 15:11 12/29/81 15:09  2422
68:
```

Since all the columns are in very predictable places, it is possible to easily extract the information desired by judicious use of the MID$ function. Also, since this is a subdirectory, there is no line showing blocks free and blocks in use. Try using this program on a volume directory. The last string read from the file will contain this information, very useful if you want to check for imminent "Disk Full" errors. Also, since the volume directory lists all the subdirectories (labeled "CAT" in the file type column), it is possible to get a full list of all the files on a volume by successively reading the individual subdirectory files. Another treat for the esoteric members of the audience is to compare the information in the hex dump with the formatted output to discover where and how SOS hides all the information about files.

New Stuff

As I promised last time, I want to go briefly into one of the most powerful new capabilities of Business Basic, the REQUEST invokable module. Normally, all access to SOS files is done through the INPUT, PRINT, READ, WRITE and GET statements of Basic. Basic interprets your desires and performs operations called "SOS calls" to do the actual work of reading and writing to physical devices. There are times, however, when the programmer needs direct access to the information which SOS has about files, and other times when certain status and control information needs to be interrogated or set. More information about what this information consists of for a particular driver can be found in the appropriate reference manual for that driver.

Of greatest interest to us now, however, is the ability to use SOS to directly read and write data to files. A single SOS FWRITE command can transfer up to 64K bytes of data to a file. Normally Basic allows writing only one variable at a time, and although it is possible to put more than one value in a single print or input statement, there are real limits on the amount of data which can be transferred at one time. This generally means that arrays of data get written using FOR-NEXT loops, adequate, but hardly a speed-burner.

To help solve this problem in situations where performance is at a premium, the REQUEST module contains two proceedures: FILREAD and FILWRITE. They are documented in the REQUEST.DOC file on the Basic disk, but for reference, here are the formats:

```
PERFORM FILREAD(%filnum,@array$,%numbytes,@count)
PERFORM FILWRITE(%filnum,@array$,%numbytes)
```

"filnum" refers to the file number you used in the OPEN statement for the file to be read or written. It can be any file which Basic is allowed to open. This includes device files like ".console" as well as disk-based text and data files. The "%" symbol in front of the "filnum" indicates that you should either use an integer variable, or put the % in front of any constant you use, to insure that an integer value is passed to the procedure. "Array$" refers to a string variable which contains the name of the array which your wish to read or write. The "@" character on the front of the string variable name instructs Basic to pass the memory address of the string, not the actual contents of the string itself. The invokable module is responsible for finding out what array name is in the string, and then locating the array in memory. The "numbytes" parameter tells the procedure how many bytes are to be read or written from the array. In the FILREAD procedure, the extra parameter "count" allows the procedure to pass back information about how many bytes were actually read, in case an EOF or other event prevented the reading of the full amount of data specified. It must be an integer variable.

One note is important here. These procedures read and write the exact contents of arrays. In the case of disk files, there is no way to read this data back once it is written, except by using the FILREAD procedure. That is, if you write an integer array to a DATA file, no type bytes are placed in the file, just the binary integer values, one after the other. The same is true for text files. Normal writes to text files convert the binary internal format to ASCII character format. If you write to a text file using FILWRITE, the exact binary data is written. You can position the file pointer using random access statements, but once a FILWRITE starts, it does not respect record boundaries. Great care must be taken if you have any ideas about mixing this kind of data with the normal contents of text and data files. I often use record 0 of the file to document the use of FILREAD and FILWRITE within an ordinary file by putting information there about the types of arrays, their location within the file, their length, etc.

Now that we've documented how it works, let's look at an example which will demonstrate how it can improve the performance of your programs.

The program below represents a "benchmark" of the time it takes to write a real and an integer array to a data file:

```
10   DIM realarray(10,100),intarray%(10,100)
20   OPEN#1,"test.request"
```

```
 30   REM fill arrays with random data
 40   FOR i=1 TO 10
 50     FOR j=1 TO 100
 60       val=RND(1)*30000:valint%=INT(val)
 70       realarray(i,j)=val:intarray%(i,j)=valint%
 80       NEXT j,i
 90   PRINT"Arrays filled."
 100   PRINT"Writing real array with FOR-NEXT."
 110   PRINT"Start time: "; TIME$;
 120   FOR i=1 TO 10:FOR j=1 TO 100
 130       WRITE#1;realarray(i,j)
 140       NEXT j,i
 150   PRINT"   Stop time: "; TIME$
 160   PRINT"Writing integer array with FOR-NEXT."
 170   PRINT"Start time: "; TIME$;
 180   FOR i=1 TO 10:FOR j=1 TO 100
 190       WRITE#1;intarray%(i,j)
 200       NEXT j,i
 210   PRINT"   Stop time: "; TIME$
 220   CLOSE
 230   END
```

As you can see, this is a relatively straightforward program which writes a 1000 element real array and a 1000 element integer array to disk. My apologies to those of you without clock chips. One of the only rewards for getting Apple III serial number 337 was that I still have a semi-functional clock. If you run this program, the timings should look something like this:

```
)RUN

Arrays filled.
Writing real array with FOR-NEXT.
Start time: 13:37:42   Stop time: 13:38:17
Writing integer array with FOR-NEXT.
Start time: 13:38:17   Stop time: 13:38:38
```

All this adds up to about 35 seconds to write the real array, and 21 seconds to write the integer array. A great deal of this time is spent in the FOR-NEXT loop, and in writing each element separately. Now let's look at the same program using FILWRITE:

```
 10   DIM realarray(10,100),intarray%(10,100)
 20   OPEN#1,"test.request"
 25   INVOKE".d1/request.inv"
 30   REM fill arrays with random data
 40   FOR i=1 TO 10
 50     FOR j=1 TO 100
```

```
60       val=RND(1)*30000:valint%=INT(val)
70       realarray(i,j)=val:intarray%(i,j)=valint%
80       NEXT j,i
90   PRINT"Arrays filled."
95   array$="realarray"
100   PRINT"Writing real array with FILWRITE"
110   PRINT"Start time: "; TIME$;
120   PERFORM filwrite(%1,@array$,%4000)
150   PRINT"   Stop time: "; TIME$
160   PRINT"Writing integer array with FILWRITE"
165   array$="intarray%"
170   PRINT"Start time: "; TIME$;
180   PERFORM filwrite(%1,@array$,%2000)
210   PRINT"   Stop time: "; TIME$
220   CLOSE
230   END
```

Notice that in the "filwrite" PERFORM statements, that %1 was used to denote the fact that we wanted to write to file number 1, and the string "array$" contained first contained the arrayname "realarray" and then "intarray%".

Also, a length of 4000 was used in the case of the real array (1000 elements at 4 bytes each) and 2000 in the case of the integer array (1000 elements at 2 bytes each). The result when this version is run is quite dramatic:

```
)RUN

Arrays filled.
Writing real array with FILWRITE
Start time: 13:54:45   Stop time: 13:54:48
Writing integer array with FILWRITE
Start time: 13:54:48   Stop time: 13:54:49
```

That's right! Approximately three seconds was required for the real array, and only one second for the integer array, between ten and twenty times faster than the previous example. Remember, though, that data written with this technique is readable only with a similar FILREAD statement, and if you ever lose track of the way in which it was written, it's tough toenails. Even with those minor difficulties, I'm sure that you'll find lots of good uses for this new invokable module.

New Stuff - Part Two

I have been promising for several months now to pass along some information about the PRINT USING capabilities of Business Basic. Rather than go into detail about every little feature, I decided to give a quick overview of the main features, and then give an example which shows off some of the power of

PRINT USING, as well as answering some questions about how to use the Long Integer data type for financial accounting applications. That's a lot to stuff into one section, but here goes:

Like most PRINT USING implementations in various dialects of Basic, Business Basic permits the printing of a list of variables according to a format described in an IMAGE statement. In fact, if you have programs in Microsoft Basic, CBASIC or most others with simple IMAGE statements, they should convert readily. It is in the extensions to these simple capabilities where Business Basic really starts to shine. The standard format is, as was said, like the following:

```
10   PRINT USING 20; first$,firstnum,secondnum%
20   IMAGE AAAAAAAAAAAAAAA,XXX,#####.##,XXX,#####
```

In the Image statement, "A" reserves a space for one alphabetic character, "X" inserts a blank space, "#" reserves a space for one numeric digit, and "." tells Basic where to align and print the decimal point in a numeric field. Therefore, the example above in line 20 is interpreted as follows:

"Print the string variable "first$" in the first 15 positions of the output record, skip 3 spaces, then print the real variable "firstnum" with 5 digits to the left of the decimal point, and 2 decimal places to the right. Then skip another 3 spaces, and print the integer variable "secondnum%" right justified in a 5 digit field."

Assuming the values "My test string" for first$, 123.443 for firstnum, and -2345 for secondnum%, the output would look like this:

```
My test string      123.44   -2345
```

Other questions, like what happens when the number or string is too big to fit, are best left to a careful reading of the Basic reference manual. Now the fun begins. Business Basic allows considerable flexibility in the way the simple example above can be expressed. For one thing, it can be simplified by placing repeat factors on the specification characters, like this:

```
20   IMAGE 15A,3X,5#.2#,3X,5#
```

Another feature is that the "image" string can be a string value replacing the line number reference in the PRINT USING statement. The following are equivalent:

```
10   PRINT USING "15A,3X,5#.2#,3X,5#";first$,firstnum,secondnum%
```

or

```
10   format$="15A,3X,5#.2#,3X,5#"
20   PRINT USING format$;first$,firstnum,secondnum%
```

It is this last variation, and the power it gives us to change the format under program control, that we will explore in depth a little later.

So far we have covered the "X" specification, called a "literal" spec, the "A" spec, called a "string" spec, and the "#" spec, called a "digit" spec. Others available include:

Literal Spec

X prints a space

/ prints a carriage return "any text" inserts literal strings in the output

---

Digit Spec

Reserves one digit, leading zeros are suppressed

& Reserves one digit or comma. Commas are inserted every 3 digits

Z Reserves one digit, leading zeros are printed

Special Numeric Specs

+ Reserves position for a sign

- Prints sign only if negative (default)

++ Prints "floating sign" in rightmost unused position

-- Prints "floating sign" only if negative

$ Reserves position for dollar sign ("$")

$$ Prints "floating dollar sign" ** Fills leading spaces with asterisks

E Prints the number in scientific or engineering notation

String Specs

A Prints string left-justified in the field

C Prints the string centered in the field

R Prints the string right-justified in the field

As you can see, these options give the programmer quite a bit of flexibility in outputing information, especially in business and scientific applications. What gives even greater flexibility is the fact that PRINT USING works with files, by

using the PRINT USING#n form of the statement, and even works with random access text files by substituting PRINT USING#n,rec.

One other feature of PRINT USING is important to mention. Many Business programmers, especially in accounting applications, must use integer arithmetic to insure "penny accuracy". i.e. no round-off errors from floating point calculations. Ordinary Basics hamper this effort, however, because PRINT USING cannot insert decimal points in integer values. Business Basic has a special function, used only in PRINT USING output lists, to solve this problem. The function is called SCALE, and can be used with any numeric value to apply a relative power of ten (decimal point shift) to the number being printed. The format looks like this:

```
SCALE(scalefactor,numericvariable)
```

For example, the following:

```
10  longnum&=12345678
20  PRINT USING "7#.2#"; SCALE(-2,longnum&)
```

would result in the output:

```
123456.78
```

To illustrate the use of these features in business applications, the following program will be used. We'll set it up to accept numbers with decimal points in them, convert them to long integers with a scale factor based on the number of places to the right of the decimal point, and then create a subroutine which can add any two scaled integers together without loss of precision. Finally, we'll set up a routine which uses SCALE and a PRINT USING spec in a string variable to print out the result with the correct number of decimal places.

First, the routine to input two numbers and do the conversion and scaling:

```
10   PRINT:INPUT"First number: ";a$
12   IF a$="" THEN END
15   GOSUB 905
17   IF errorcode THEN PRINT"Range exceeded, try again.":GOTO 10
20   scale.first%=scale%:first&=a&
25   INPUT"Second number: ";a$
30   GOSUB 905
32   IF errorcode THEN PRINT"Range exceeded, try again.":GOTO 25
35   scale.second%=scale%:second&=a&
40   PRINT USING 45;first&,scale.first%
45   IMAGE " first value= ",20#,"  scale factor= ",3#
50   PRINT USING 55;second&,scale.second%
55   IMAGE "second value= ",20#,"  scale factor= ",3#
60   END
899   REM
```

```
 900   REM subroutine to convert input to long integer plus scale
 905   errorcode=0:ON ERR errorcode= ERR:OFF ERR:RETURN
 915   x=INSTR(a$,".")
 920   IF x=0 THEN a&=CONV&(a$):scale%=0:OFF ERR:RETURN
 925   scale%=-(LEN(a$)-x)
 930   a$=MID$(a$,1,x-1)+MID$(a$,x+1)
 935   a&=CONV&(a$):OFF ERR:RETURN
```

The subroutine is really pretty simple. It uses the trusty INSTR function in line 915 to look for a decimal point in the input string. If none is found (i.e. number is an integer), then the string is converted to a long integer, the scale factor is set to zero, and a return is taken. Note that conversion errors (overflow, etc.) are handled by the ON ERR statement, which passes back the errorcode to the calling program. If a decimal point is found, the scale factor is set to the number of digit positions from the point to the end of the string (line 925) and line 930 and 935 scrunch out the decimal point and convert the resulting integer to a long integer value. Once back in the input routine, the errorcode flag is checked, and if everything is ok, some simple PRINT USING statements print out the result for comparison. It should be noted that these routines are not bulletproof, but were deliberately kept simple to illustrate the major points involved.

Now that we have long integer representations of these decimal numbers, with appropriate scale factors, it is possible to create a routine which will perform arithmetic on them, even though they may have different scale factors. The following routine will illustrate addition:

```
 1000   REM add a& and b& and return result in sum&
 1001   REM use scalea% and scaleb% to return scalesum%
 1005   errorcode=0:ON ERR errorcode= ERR:OFF ERR:RETURN
 1010   IF scalea%=scaleb% THEN sum&=a&+b&:scalesum%=scalea%:OFF
ERR:RETURN
 1020   IF scalea%>scaleb% THEN 1070
 1030   factor%=scaleb%-scalea%
 1040   b&=b&*CONV&(10^factor%)
 1050   sum&=a&+b&:scalesum%=scalea%:OFF ERR:RETURN
 1070   factor%=scalea%-scaleb%
 1080   a&=a&*CONV&(10^factor%)
 1090   sum&=a&+b&:scalesum%=scaleb%:OFF ERR:RETURN
```

The first thing checked for is if the two numbers have the same scale factor. If so, then simple addition is all that is required, and scalesum% (the resulting scale factor from the operation) is set to the common scale. If the scale factors are unequal, then the two scale factors must be adjusted to be the same by multiplying the one with the larger scale by the power of ten required to make them equal in scale. An example would help clarify:

```
Initial number         Integer value      Scale factor

 12345.6789              123456789            -4
   98765.43                9876543            -2
```

Obviously, just adding the two integers will produce meaningless results. But multiplying the second number by 100 and adjusting the scale factor correspondingly to -4 will make it possible to directly add them. The situation now looks like this:

```
New format             Integer value      Scale factor

12345.6789               123456789            -4
98765.4300               987654300            -4
```

The sum of the integer values is 1111111089 and, after applying the scale factor of -4, the result is 111111.1089. You should realize that most floating point Basics, no matter how many digits they allow in "Double Precision" mode, have extreme difficulty with these types of problems. The reasons are complex, but they have to do with the fact that there are some decimal fractions which cannot be represented exactly with a binary floating point ("real") number. This leads to potential loss of precision in the last decimal place, rendering the answer inaccurate. While one place out of ten or fifteen might not be critical in an empirical scientific calculation, accountants are fussy about all the pennies (or in the example above, tenths of mils) adding up exactly. NOte also that scale factors can just as easily be positive. That is, 567890000 could be represented as 56789 with a scale factor of 4. The principles of addition would work exactly the same as in the example with decimal fractions.

With the techniques described above, I think you can now figure out the way the subroutine works. One final note, though. In line 1040 and 1080 we use an expression "10^factor%" to represent the power of ten to be multiplied by the long integer value. Mixed mode expressions are not allowed between long integers and other data types, so the CONV& function was used first to convert the power of ten expression to a long integer.

Now that we have a subroutine which will correctly add two scaled numbers, we can put it into our previous input program. The combination looks like this:

```
 5   PRINT"Test of extended precision add routines":PRINT
 10   PRINT:INPUT"First number: ";a$
 12   IF a$="" THEN END
 15   GOSUB 905
 17   IF errorcode THEN PRINT"Range exceeded, try again.":GOTO 10
 20   scale.first%=scale%:first&=a&
 25   INPUT"Second number: ";a$
 30   GOSUB 905
 32   IF errorcode THEN PRINT"Range exceeded, try again.":GOTO 25
```

```
 35   scale.second%=scale%:second&=a&
 40   PRINT USING 45;first&,scale.first%
 45   IMAGE " first value= ",20#,"  scale factor= ",3#
 50   PRINT USING 55;second&,scale.second%
 55   IMAGE "second value= ",20#,"  scale factor= ",3#
 60   scalea%=scale.first%:scaleb%=scale.second%
 65   a&=first&:b&=second&
 70   GOSUB 1010
 72   IF errorcode THEN PRINT"Range of precision exceeded, try
again.":GOTO 10
 75   PRINT"sum= ";sum&;"   scale factor= ";scalesum%
 105   GOTO 10
 899   REM
 900   REM subroutine to convert input to long integer plus scale
 905   errorcode=0:ON ERR errorcode= ERR:OFF ERR:RETURN
 915   x=INSTR(a$,".")
 920   IF x=0 THEN a&=CONV&(a$):scale%=0:OFF ERR:RETURN
 925   scale%=-(LEN(a$)-x)
 930   a$=MID$(a$,1,x-1)+MID$(a$,x+1)
 935   a&=CONV&(a$):OFF ERR:RETURN
 999   REM
 1000   REM add a& and b& and return result in sum&
 1001   REM use scalea% and scaleb% to return scalesum%
 1005   errorcode=0:ON ERR errorcode= ERR:OFF ERR:RETURN
 1010   IF scalea%=scaleb% THEN sum&=a&+b&:scalesum%=scalea%:OFF
ERR:RETURN
 1020   IF scalea%>scaleb% THEN 1070
 1030   factor%=scaleb%-scalea%
 1040   b&=b&*CONV&(10^factor%)
 1050   sum&=a&+b&:scalesum%=scalea%:OFF ERR:RETURN
 1070   factor%=scalea%-scaleb%
 1080   a&=a&*CONV&(10^factor%)
 1090   sum&=a&+b&:scalesum%=scaleb%:OFF ERR:RETURN
```

Notice that in addition to adding the subroutine at line 1000, I have added some code at 60 through 105 to set up the call to the subroutine and then print out the results. This is all fine, but this was supposed to be an exercise in advanced uses of the PRINT USING statement. An ideal use of Print Using here would be to print out the results of the addition, with the decimal point in the proper place. But, since our answers can range from 19 digits to the left of the decimal place to 19 digits to the right, and only a total of 32 positions are allowed in a single numeric IMAGE field, it is not possible to create a single format which will handle all possible variations. Here's where Business Basic's ability to have variable format definitions really comes in handy. The following routine can be

added to the program above to print the result correctly, no matter what the scale factor:

```
80   x%=LEN(CONV$(sum&)):neg%=CONV%(sum&<0)
85   IF x%+scalesum%-neg%<=0 THEN
form$="2#":ELSE:form$=CONV$(x%+scalesum%)+"#"
90   IF scalesum%>=0 THEN 97
95   form$=form$+"."+CONV$(ABS(scalesum%))+"#"
97   PRINT"scaled result of sum: ";
100   PRINT USING form$; SCALE(scalesum%,sum&)
```

Line 80 gets the length of the number to be printed in x% and neg% is a flag to tell if the number is negative (the minus sign will require an extra position in the output). Line 85 uses this information, including the value of scalesum%, to figure out how many positions are needed to the left of the decimal point. Line 85 then creates form$, the output format specification, to match. Line 90 checks to see if scalesum% is positive (i.e. value is a true integer). If so, its finished. Otherwise, line 95 creates the rest of the format spec by including the proper number of positions to the right of the decimal point. Line 97 and 100 then print out the long integer using the SCALE function to properly place the decimal point.

Voila! This routine should give exactly correct answers over its range of values. One thing you might want to add to help in tracing what the program is doing is to print out the value of form$ along with the result in line 100. Also, I leave for your personal entertainment the creation of subroutines for subtraction and multiplication. Division can be done using a combination of the DIV and MOD operators, but you will become embroiled in what to do about rounding off the results of certain divisions. Multiplication has the virtue of being exactly correct within the possible range of values.

Well, what an exercise! When I made up the list of topics last time, I had no idea it would produce such a wordy article. It is with only a little regret, therefore, that I announce that I am not going to announce the topic of next month's article. It should be a goodie, however, because there is a lot left to explore in our favorite Basic. Another thought comes to mind, too. If you've got a favorite subject you'd like to see examined, why not write in and suggest it. I'd like to make this column as useful as possible to those of you working with the language and creating applications. Until March, then...

Exploring Business Basic, Part VII

Last month we explored some of the Business Basic's unique formatted output and arithmetic capabilities. There is a good more to say on those topics, but such exposition will be left to some future month. This month we will undertake a journey through some of the thickest jungles found in the Apple III, the Infamous .GRAFIX driver and its faithful Indian companion, BGRAF.INV (The preceding collection of mixed metaphors was just a sample of what some enterprising explorers have encountered on their own trips).

The new Business Basic manual (which everyone who purchased Basic should have received by now), contains a sixty page section in Volume Two which describes the programming possible with the BGRAF.INV invokable module. In addition, the Standard Device Drivers Manual contains a section on .GRAFIX. Rather than repeat all of that material, this column will briefly describe the functions of BGRAF.INV and then take up a subject which is not mentioned at all, HOW TO DRAW A CIRCLE. Drawing a circle may sound easy, but given the fact that BGRAF only allows dots and lines, and given that none of the graphics modes have equal horizontal and vertical resolution, and given that monitors distort images because of "aspect ratio" differences, we will see that drawing a circle of arbitrary radius with an arbitrary center that actually looks like a circle and doesn't take forever to finish is non-trivial. Non-trivial is a favorite word of mathematicians and engineers, principally because it allows them to assert that a task is difficult, but doesn't require that they figure out how difficult.

Well, enough cheap shots at mathematicians. We will discover later that some handy mathematical principles will serve us well in our quest for the perfect circle.

The BGRAF Invokable Module

As has been discussed before, Business Basic is almost infinitely extensible by the use of Invokable assembly language routines. These routines can be loaded into memory only when needed, and have the effect of adding extra commands to the language. Furthermore, as many "Invokables" as will fit into memory can be used at once, with Apple III's SOS operating system responsible for making sure that there are no conflicts. BGRAF.INV is one of the most useful of the invokable modules. It is supplied on the Business Basic program disk, and is loaded into memory with the command:

```
INVOKE "bgraf.inv"
```

Those of you who have Apple]['s, or who have used Applesoft in emulation mode, know that there are several commands in that language to manipulate color graphics. Among these are GR and HGR, COLOR and HCOLOR, PLOT and HPLOT which permit initializing graphics modes, changing colors, and plotting points and lines. In addition, Applesoft has special commands which permit the manipulation of "shapes" based on special tables which describe the bit pattern of the image.

The BGRAF invokable has commands for all these capabilities, and a great deal more. Unlike Applesoft, which has a fixed high-resolution page for drawing, the Apple III graphics modes permit plotting points within a range of -32768 to 32767. The concept of a Viewport (like the Window in text mode) is what defines which dots actually get plotted on the screen. Only the dots within the current viewport are actually plotted, and the Viewport is limited to the maximum resolution of whatever graphics mode is selected. We'll see in a minute how handy this is, because it permits plotting generally without regard to whether the physical screen limits are exceeded. Exceeding the valid range in an Applesoft program causes an error. In addition, setting the viewport to an area smaller than the physical screen permits us to draw without worrying about overwriting other areas outside the viewport. To keep track of where the plotting operations are to take place, an "invisible" cursor is maintained, to which all draw and print operations relate.

There are two capabilities of the Apple III graphics driver which are not well understood but can be extremely powerful. These are the infamous Color Table and Transfer option. Used properly, they can save an incredible amount of programmer effort. The color table allows you to set the priority of a given color. That is, imagine that your want to draw some blue lines on a screen which contains some yellow squares. Furthermore, you do not want to cross the yellow squares (in effect, you want to draw the line "behind" the squares). By setting up the color table properly, the graphics driver will automatically change any blue dots you plot to yellow if you try to plot them over a yellow dot. In any other system, your program would have to check the color of each dot before plotting, thus grinding everything to a virtual halt. Imagine what this capability might mean with shapes of various colors which you might want to animate over a background!

One additional capability in Apple III graphics is really convenient. At any time you can print text directly to the GRAFIX driver and it will be written at the current dot position. Since you can also change the definition of the character set with the NEWFONT procedure, hi-res animation tricks such as are found in the Apple]["DOS Toolkit" package are essentially built-in!

To give you an idea of the functions of the BGRAF module, the table below lists the commands available, along with a brief description. Remember that to use

these in Basic, the module must first be INVOKEd, and the word PERFORM is prefixed to each command.

INITGRAFIX - Initializes the viewport, cursor position, color table and transfer options

GRAFIXMODE - Sets the current graphics mode. The four modes are:

0 - 280 x 192 Black and White

1 - 280 x 192 Color (16 colors with limitations)

2 - 560 x 192 Black and White

3 - 140 x 192 Color (16 colors with no limitations)

GRAFIXON - Displays the current graphics screen

VIEWPORT - Sets the boundaries for graphics operations

PENCOLOR - Sets the color of the "pen" for draws, plots or characters

FILLCOLOR - Sets the background color for filling and erasing

FILLPORT - Fills the current viewport with the fill color

MOVETO - Moves the cursor to a specified point

MOVEREL - Moves the cursor relative to the current point

DOTAT - Plots a point at a specified point

DOTREL - Plots a poiint relative to the current point

LINETO - Draws a line from the current point to a specified point

LINEREL - Draws a line to a point relative to the current point

XYCOLOR - Function that returns the color of a specified point

XLOC,YLOC - Functions that return the current position of the cursor

NEWFONT - Defines a new character font for printing text on the screen

SYSFONT - Restores the default system font

GSAVE - Saves a graphics screen to disk as a PIC file

GLOAD - Loads a PIC file from disk to the current graphics screen

RELEASE - Gives graphics memory back to Basic

Quite a collection of goodies, right?

As was said earlier, there is really too much here for one article. Indeed, a whole book could be written about the Apple III graphics. Rather than tackle that task, let's start with something seemingly simple. As you noted from reading the list of functions above, the major missing component is anything to do with curves. Rather than throwing you a curve (groan!), we'll try drawing some.

Getting Around in Business Basic

To start, some quick math is required. You can think of the Apple III screen as a coordinate system, with X and Y locations depending on the mode. In all cases, Y (vertical) values are displayable between 0 (bottom) and 191 (top). X (horizontal) values range between 0 and 139 (lowest resolution) up to 0 to 559 (highest). Circles are nothing more than a set of points with a common attribute, namely they are of equal distance from a single point, called (surprisingly enough) the center. There are formulas for determining the points which lie on a circle, generally derived from the formula below:

```
2     2    2
X  +  Y  = R
```

This formula works for circles starting at a center of 0,0, but since we want to draw circles anywhere, and since the general form of the circle equation is more difficult to solve, we will rely on another fact about circles, namely that the trigonometric functions SINe and COSine define X and Y values for the Unit circle, and it is possible to obtain values for any circle by multiplying these values by the radius and adding the center coordinates. That is, to find a point on a circle of radius 30 at an angle of 30 degrees from horizontal, when the center is at X=70 and Y=96 these formula can be used:

```
X = COS(30 degrees)*30+70
Y = SIN(30 degrees)*30+96
```

This simple formula suggests that we might be ready to write a program:

```
 10   OPEN#1,".grafix"                  Open the graphics driver
 20   INVOKE"bgraf.inv"                 Load BGRAF into memory
 30   PERFORM initgrafix                Initialize the graphics screen
 40   PERFORM grafixmode(%3,%1)         Set mode 3 (16 color)
 45   INPUT"step value: ";stepval       Ask for an increment for
plotting
 50   PERFORM grafixon                  Turn on the graphics display
 60   PERFORM pencolor(%13)             Set the color for drawing
(Yellow)
 70   PERFORM fillcolor(%3)             Set the background color
(Purple)
 80   PERFORM fillport                  Fill the viewport with Purple
 100   FOR i=0 TO 6.28 STEP stepval     Step around the circle
(2pi=6.28)
```

```
 110     x=COS(i)*30*(140/192)+70       Calculate x  (center=70,
radius=30)
 120     y=SIN(i)*30+96                 Calculate y  (center at y=96)
 130     PERFORM dotat(%x,%y)           Plot the resulting dot location
 140     NEXT i
 150   INPUT a$                         Pause when finished
 160   TEXT                             Switch to Text mode
 170   PERFORM release:PERFORM release  Clean up the graphics memory
 180   PERFORM release:INVOKE:CLOSE
 190   END
```

This is a relatively straightforward program, except to note that the trig functions (SIN, COS) work in radians, of which there are 2 x pi in a full circle. That value is approximately 6.2832, which is further approximated in line 100. One other thing of note: Since this graphics mode is not "square", some adjustment must be made for the fact that there are more points proportionately in one axis that another. For simplicity, we have scaled the X-axis value (since that is the only one that varies in the different modes) by multiplying by the constant 140/192, the ratio of horizontal to vertical dots. This is done in line 110. The other factor in line 110 and 120 is the constant 30, which represents the radius. Note also that line 170 cleans up the graphics memory and closes the driver. This is VERY important. If you don't release the memory, it will stay around, unusable by BASIC. Also, doing the INVOKE in line 180 removes the BGRAF module from memory. If you have other invokables normally resident, you should delete this statement.

Run this program several times, with different values for the step. In addition to being pretty slow, you will notice that it takes a step size of about .1 to draw a good circle. You probably also noticed that this program can't draw a "good" circle. Depending on the "aspect ratio" of your monitor, the circle will probably look like a flattened circle, that is, an ellipse. This is the result of the fact that all monitors differ in the relationship between horizontal and vertical resolution and size. We will see a little later that this is an easy problem to correct.

As was just mentioned, this routines suffers from being very slow. The main problem stems from the fact that it takes a large number of dots to create a circle, and that number of dots translates into a large number of steps to draw a circle. In the example above, it took approximately 64 dots to draw a filled in circle of radius 30 units. Had we tried to draw a larger circle, or had we used a higher resolution mode, the problem would have been worse. The solution to this problem lies in understanding the real nature of the task at hand.

Mathematics and Mathematical Physics is sometimes called the realm of the "perfect". Energy is truly exactly related to mass times the speed of light squared (Einstein's famous formula). However, in the world of measured events, nothing is ever exact and perfect. The same is true of circles. Geometry allows

us to dream of perfect circles, but the realities of trying to draw one (especially freehand!) are such that we are willing to settle for reasonably good representations as long as they are not too lumpy. In fact, the resolution of any graphics screen, no matter how good, is a far cry from the perfection of a "real" circle. Therefore, when we set out to draw a circle on the screen, we should first ask "how good a circle do you want?"

A quick lesson from geometry will help with the answer. As you may know, a circle can be approximated as a polygon (a figure with many sides). The more sides the polygon has, the more it looks like a circle. Since the resolution of the graphics screen is limited, at some reasonable point a polygon will be indistinguishable from a circle. The advantage of this approach is that the Apple III has graphics commands available to draw lines. Since a polygon has sides which are all straight lines, we can use the line drawing commands to represent a circle, finding the number of sides in each resolution which make reasonable looking circles. The number of sides necessary to make "good" circles will also vary according to the radius of the circle, since large circles will be more likely to show the lines as straight segments. The following example will let you play with the number of sides necessary to make good circles, and also, to experiment with the aspect ratio which is correct for your monitor. You should modify this program to try other graphics modes, and see how the different resolutions affect the results.

```
10   OPEN#1,".grafix"
20   INVOKE".d1/bgraf.inv"
30   PERFORM initgrafix
40   PERFORM grafixmode(%3,%1)
45   INPUT"step value: ";stepval
46   IF stepval<=0 THEN 170
48   INPUT"aspect ratio: ";aspect
49   aspect=1/aspect
50   PERFORM grafixon
60   PERFORM pencolor(%13)
70   PERFORM fillcolor(%3)
80   PERFORM fillport
85   scale=140/192
90   PERFORM moveto(%30*scale*aspect+70,%96)
100   FOR i=stepval TO 6.28 STEP stepval
110     x=COS(i)*30*scale*aspect+70
120     y=SIN(i)*30+96
130     PERFORM lineto(%x,%y)
140     NEXT i
150   INPUT a$
160   TEXT
165   GOTO 45
```

```
170   PERFORM release:PERFORM release:PERFORM release
180   INVOKE:CLOSE
190   END
```

Several new things are done in this example. First, notice that we invert the aspect ratio because we are adjusting the x-axis only. Further, we have named the ratio between the x and y resolution "scale" for use in the repetitive calculations. Next, because we are drawing lines this time, the program uses the "moveto" proceedure to move the graphics cursor to the initial point on the circle (in this case, the horizontal point to the right of the center (origin) of the circle). Once a starting point is established, subsequent "lineto" commands will draw the circle as a series of line segments. Try experimenting with widely varying numbers of steps, from 1 to .02 as an example. You will find that at some point the circle looks the same, no matter how many line segments make it up. By choosing the least number of steps which still produce a decent circle, you can speed up the drawing considerably. Don't forget to experiment with values for aspect ratio as well. For the Monitor III, a value of 1.3 usually works pretty well. Try several until you are happy with the results.

The next technique for speeding up this routine is even more interesting. Notice that we keep calculating the SINE and COSINE of each angle, no matter how many times we run the program. Furthermore, it should be noticeable that it is possible to draw a circle of any reasonable radius by just varying the multiplication factor. Further, it is obvious that steps of less than .05 for any reasonable radius do not produce "better" circles. All that suggests the following "enhanced" version of the program:

```
10   OPEN#1,".grafix"
20   INVOKE".d1/bgraf.inv"
25   DIM xcos(63),ysin(63)
26   FOR i=0 TO 63:xcos(i)=COS(i/10):ysin(i)=SIN(i/10):NEXT i
30   PERFORM initgrafix
40   PERFORM grafixmode(%3,%1)
46   INPUT"aspect ratio: ";aspect
47   aspect=1/aspect
48   INPUT"radius: ";r
49   IF r<=0 THEN 170
50   PERFORM grafixon
60   PERFORM pencolor(%13)
70   PERFORM fillcolor(%3)
80   PERFORM fillport
85   scale=140/192
90   PERFORM moveto(%r*scale*aspect+70,%96)
100   FOR i=1 TO 63
110     x=xcos(i)*r*scale*aspect+70
120     y=ysin(i)*r+96
```

```
130     PERFORM lineto(%x,%y)
140     NEXT i
150   INPUT a$
160   TEXT
165   GOTO 48
170   PERFORM release:PERFORM release:PERFORM release
180   INVOKE:CLOSE
190   END
```

Notice this time that two arrays have been set up, both with 64 values each. Rather than recalculate the SIN and COS function, they are done once at the begining and stored for later use in line 110 and 120. If you intend to do a great deal of this kind of work, or if you want to expand the number of steps significantly, you may want to create another program which calculates the values and writes them to a Data file. Then your circle program could simply read the values in at the beginning.

Notice too that these routines are essentially identical to the previous ones, except that this time you may experiment with circles of different radius. After experimenting with this routine, you should see that a general purpose routine can be written which will satisfy all circumstances. For most practicality, this can be expressed as a subroutine, with the variables being the circle diameter, the mode, the scale factor and the center coordinates. The result could look like this:

```
10   OPEN#1,".grafix"
20   INVOKE".d1/bgraf.inv"
25   DIM xcos(126),ysin(126),xdot(3)
26   FOR i=0 TO 126:xcos(i)=COS(i/20):ysin(i)=SIN(i/20):NEXT i
27   xdot(0)=280:xdot(1)=280:xdot(2)=560:xdot(3)=140
30   PERFORM initgrafix
35   INPUT"Mode: ";mode
37   IF mode<=0 THEN 180
40   PERFORM grafixmode(%mode,%1)
50   INPUT"pencolor,fillcolor: ";pen,fill
60   PERFORM pencolor(%pen)
70   PERFORM fillcolor(%fill)
75   INPUT"clear screen? ";a$
80   a$=MID$(a$,1,1):IF a$="y" OR a$="Y" THEN PERFORM fillport
81   INPUT"radius: ";r
82   aratio=1.3
83   xcen=xdot(mode)/2:ycen=96
85   PERFORM grafixon
87   scalefac=(1/aratio)*(xdot(mode)/192)
90   GOSUB 900
150   PERFORM moveto(%0,%8)
```

```
160   PRINT#1;"Press RETURN:";
165   INPUT"";a$
170   TEXT
175   GOTO 75
180   PERFORM release:PERFORM release:PERFORM release
190   INVOKE:CLOSE
200   END
900   xscale=r*scalefac
905   xcen=xcen+.5:ycen=ycen+.5
907   density=(mode=2)+2*(mode<2)+3*(mode=3)
910   firstx=xcos(0)*xscale+xcen
915   PERFORM moveto(%firstx,%ycen)
920   stepamt=INT(20*(5-density)/r)+density
930   IF stepamt>6 THEN stepamt=6
940   FOR i=1 TO 126 STEP stepamt
950     PERFORM lineto(%(xcos(i)*xscale+xcen),%(ysin(i)*r+ycen))
960     NEXT i
970   PERFORM lineto(%firstx,%ycen)
980   RETURN
```

This version is considerably enhanced. Notice that we have doubled the number of points which can be used, as well as introducing the ability to change modes and colors. Since we now have different possibilities for the mode, we introduce the "xdot" array, which contains the horizontal dot density required to figure the center and the scale factor. The actual drawing routine is now is a subroutine at line 900, in such a form that you could incorporate it into other programs.

To speed up the subroutine for the various graphics modes, the new concept of "density" is introduced. This is a factor which varies between one and three depending on whether the horizontal resolution is 140, 280 or 560 (modes 3, 0 and 1, and 2 respectively). Note the use of the logical statements in line 907 to replace a lot of IF and assignment statements. Be sure you work through that statement in your mind to assure yourself that the assignments work as intended.

The other thing of note is that the density factor is used in line 920 to calculate a reasonable step value for drawing the circle. You might want to substitute a few values to see just how this works. Line 930 makes sure that a reasonable number of steps are used, even if the circle is extremely small. Notice also that in line 905 the value .5 is added to the center coordinates. This has the effect of rounding the values when they are passed to the "lineto" proceedure in line 950, insuring more accurate plotting. Another new feature is that in line 970 an additional "lineto" is added to draw a line back to the original point. This insures that if the step value is such that the circle is not fully completed, then the last point drawn will be connected to the beginning point.

Line 160 introduces another new concept. By simply printing to the .GRAFIX driver file, you may write text on any graphics screen. Furthermore, the text can begin on any dot boundary anywhere on the screen. That requires some pretty tricky software on the Apple][, but is a built-in feature of the Apple III graphics modes!

Many more enhancements could be added to this program, but instead of going on and on, here's an example of how the program and routines could be modified to draw circle segments (arcs) and "pie slices", using essentially the same techniques. The new program looks like this:

```
3   REM arc draw subroutine
10   GOSUB 1000:REM initialize
20   PRINT"Arc drawer program"
35   INPUT"Graphics mode: ";mode$
36   IF mode$="" THEN 180
37   mode=CONV(mode$)
40   PERFORM grafixmode(%mode,%1)
50   INPUT"pencolor,fillcolor: ";pen,fill
52   draw.radius=0
55   INPUT"draw the radii? ";a$
56   a$=MID$(a$,1,1):IF a$="y" OR a$="Y" THEN draw.radius=1
60   PERFORM pencolor(%pen)
70   PERFORM fillcolor(%fill)
75   INPUT"clear screen? ";a$
80   a$=MID$(a$,1,1):IF a$="y" OR a$="Y" THEN PERFORM fillport
82   horiz=xdot(mode)/192
85   scalefac=(1/aratio)*horiz
87   PERFORM grafixon
88   FOR loop=1 TO 25
90     r=INT(50*RND(1)+30)
91     xcen=INT(192*horiz*RND(1))
92     ycen=INT(192*RND(1))
93     start.rad=3.14*RND(1):end.rad=start.rad+3*RND(1)
95     GOSUB 1100
100     NEXT loop
150   PERFORM moveto(%0,%8)
160   PRINT#1;"Press RETURN:";
165   INPUT"";a$
170   TEXT:GOTO 35
180   PERFORM release:PERFORM release:PERFORM release
190   CLOSE:INVOKE
200   END
1000   OPEN#1,".grafix"
1010   INVOKE".d1/bgraf.inv"
```

```
 1020   DIM xcos(126),ysin(126),xdot(3)
 1030   FOR i=0 TO 126:xcos(i)=COS(i/20):ysin(i)=SIN(i/20):NEXT i
 1040   xdot(0)=280:xdot(1)=280:xdot(2)=560:xdot(3)=140
 1050   aratio=1.3
 1060   PERFORM initgrafix
 1070   RETURN
 1094   REM       r=radius, scalefac=aspect ratio * relative density
 1095   REM       xcen= x coordinate of center
 1096   REM       ycen= y coordinate of center
 1097   REM       start.rad= starting point of arc in radians
 1098   REM       end.rad= ending point of arc in radians
 1099   REM       draw.radius=1 means draw the radius lines to the
endpoints
 1100   xscale=r*scalefac
 1105   xcen=xcen+.5:ycen=ycen+.5
 1110   density=(mode=2)+2*(mode<2)+3*(mode=3)
 1115   IF draw.radius THEN PERFORM moveto(%xcen,%ycen):PERFORM
lineto(%(COS(st
        art.rad)*xscale+xcen),%(SIN(start.rad)*r+ycen)):ELSE:PERFORM
moveto(%(C
        OS(start.rad)*xscale+xcen),%(SIN(start.rad)*r+ycen))
 1120   stepamt=INT(20*(5-density)/r)+density
 1130   IF stepamt>6 THEN stepamt=6
 1140   FOR i=INT(start.rad*20+.5) TO end.rad*20 STEP stepamt
 1150     PERFORM lineto(%(xcos(i)*xscale+xcen),%(ysin(i)*r+ycen))
 1160     NEXT i
 1170   PERFORM lineto(%(COS(end.rad)*xscale+xcen),%
(SIN(end.rad)*r+ycen))
 1175   IF draw.radius THEN PERFORM lineto(%xcen,%ycen)
 1180   RETURN
```

This program is set up to use lines 88 through 100 to create random centers, radii and arc lengths (in radians) and to use the subroutine at line 1100 to draw the resulting arcs. Between this routine and the one above to draw circles, you should be able to do most of the interesting tasks in graphics. Hopefully, these routines will also give you ideas on solving whatever other specific projects you might want to tackle.

Normally, when you run the "arc" program above, you will get some arcs which are partially off the screen. Notice that the .GRAFIX driver handles this perfectly, because it treats its graphics area as a space of points from -32768 to 32767, with the screen as a window into the total space. This saves immeasurable amounts of bounds checking within programs, which usually ends up slowing down the drawing. Additionally, as was mentioned at the top, the graphics

window can be set to anywhere on the screen, with any values outside the window automatically clipped.

There are a thousand more topics to be covered in exploring the graphics capabilities of the Apple III. Next month we will tackle a few biggies, "area fill" (especially for the circles and arcs we have been drawing), and the whole area of user-definable character sets. With luck, we'll get to some animation examples. Until then, dig into your device driver manual documentation on .GRAFIX and the writeup on BGRAF.INV in the Business Basic Manual. There's a whole world inside this system!

Exploring Business Basic, Part VII

Greetings, Basic fans. This month's column will be long on content and somewhat short on the usual herbage and explanations. This is due for the most part to the somewhat detailed interest that last month's column stirred up. As the faithful among you will recall, we began a fairly simple discourse on the graphics capability of the Apple III and the specific workings of the .GRAFIX driver and the BGRAF invokable module. If you haven't read that missive, I really suggest you get a copy before tackling the treatise below. If that's not possible, then getting a firm grip on your Basic and Standard Device Drivers manual will probably do the trick.

Last month's article had as its main feature a program to efficiently draw circles and arcs using the line-drawing capability of BGRAF. After some examination, (and some comments!) that routine could use some tweaking. To demonstrate what can be done, and to bring you somewhat in sync with last month, here are the initialization and circle draw subroutines:

```
894   REM circle draw subroutine
895   REM       r=radius, scalefac=aspect ratio * relative density
896   REM       xcen= x coordinate of center
897   REM       ycen= y coordinate of center
900   xscale=r*scalefac
905   xval=xcen+.5:yval=ycen+.5
907   density=(mode=2)+2*(mode<2)+3*(mode=3)
910   firstx=xcos(0)*xscale+xval
915   PERFORM moveto(%firstx,%yval)
920   stepamt=INT(20*(5-density)/r)+density
930   IF stepamt>6 THEN stepamt=6
940   FOR i=stepamt TO 119 STEP stepamt
950     PERFORM lineto(%(xcos(i)*xscale+xval),%(ysin(i)*r+yval))
960     NEXT i
970   PERFORM lineto(%firstx,%yval)
980   RETURN
995   REM Initialize graphics and tables
996   REM
1000   OPEN#1,".grafix"
1010   INVOKE".d1/bgraf.inv"
1020   DIM xcos(119),ysin(119),xdot(3),srch%(20,3)
1025   val=6.2832/120
1030   FOR i=0 TO 119:xcos(i)=COS(val*i):ysin(i)=SIN(val*i):NEXT i
1040   xdot(0)=280:xdot(1)=280:xdot(2)=560:xdot(3)=140
1050   aratio=1.3
1055   zip%=1
```

```
1060   PERFORM initgrafix
1070   RETURN
```

Those readers from last time will notice that to improve the symmetry of the circles, the cosine and sine tables (line 1030) have been adjusted to 120 points, which happens to divide each quadrant of the circle up into equal parts (given a maximum stepsize of 6, as used in line 930).

Rather than go into any further detail about these routines, we'll plunge into the material, which will make good use of this stuff later.

The major input from last time was to take the skeleton program and make it into something useful which would show off more of the graphics capabilities of the Apple III. One of the most reasonable approaches would be to expand the set of functions in the program, and build a beginning graphics editor, which you could then expand to your heart's content. The actual functions of such a program are reasonably easy to define. We have already implemented circles, and to that we can add lines, points, rectangles, text, the ability to fill in areas easily, the ability to erase objects and points, and finally, the ability to store images on disk for later recall. To this list you can add lots of other features of your own design, within the framework that will be described. The key to all this is the power of the BGRAF module. The benefits (an important word in marketing parlance) range from drawing up organization charts to creating interesting cartoons. So lets get to it!

The first thing needed for a screen editor is some way to locate the cursor. Since it will be difficult to distinguish the cursor from any arbitrary dot on the screen, it will be convenient for it to blink. To do that, we need a way to alternate the colors of the dot which will be our cursor, while waiting for input from the user. The interrupt capabilities to the Apple III really come in handy here, because Basic implements the "any keypress" Event within SOS. This causes an interupt to occur in Basic which your program can react to. The following short program will illustrate:

```
1   ON KBD GOTO 10
5   PRINT".";
6   GOTO 5
10   OFF KBD
30   IF KBD=13 THEN STOP
40   PRINT KBD
45   ON KBD GOTO 10
50   RETURN
```

This program will print lots of dots, stopping only to print out the ASCII value of the key you press on the keyboard. If you are not familiar with how this works, check the section in the manual that describes the ON KBD statement. This one statement is going to make our graphics editor really easy to implement.

As was said, the idea for a blinking cursor would be to alternate colors and print a succession of these alternating dots to the screen at the cursor location. There are lots of possiblities on what color is at a given screen location, and we want to move the cursor freely without worrying about destroying the image. To do this efficiently, we'll use a feature in the .GRAFIX driver called the "transfer option". This option is really eight options in one, but we'll use Inverse Replace, which will alternate colors each time a dot is plotted. Combining that with the ON KBD statement and adding the initialization section looks something like this:

```
 10   GOSUB 1000:REM initialize
 20   HOME:PRINT"Design program"
 35   INPUT"Graphics mode: ";mode$
 36   IF mode$="" THEN 180
 37   mode=CONV(mode$)
 40   PERFORM grafixmode(%mode,%1)
 50   INPUT"pencolor,fillcolor: ";pen,fill
 60   PERFORM pencolor(%pen)
 70   PERFORM fillcolor(%fill)
 75   INPUT"clear screen? ";a$
 80   a$=MID$(a$,1,1):IF a$="y" OR a$="Y" THEN PERFORM fillport
 82   horiz=xdot(mode)/192
 85   scalefac=(1/aratio)*horiz
 87   PERFORM grafixon
 89   ON KBD GOTO 300
 90   PERFORM xfroption(%4)
 91   savecolor%= EXFN%.xycolor
 92   color%= EXFN%.xycolor:PERFORM pencolor(%color%):PERFORM
dotrel(%0,%0):GOTO 92
```

Lines 10 through 87 call the initialization subroutine, prompt for some key values, and then turn on the graphics screen. The really interesting stuff starts at line 89 where the ON KBD is set up to go to line 300 on any keypress. Line 90 sets up the transfer option (see your manual for more details on what this does), and line 91 saves the current color for restoring later. Line 92 is a loop which can only be interrupted by a keypress. It picks up the current color, sets the pen to that color, and then plots the dot. Because of the transfer option, the actual color plotted will be a logical alternate color to the original one. Plotting the alternate color through the same option restores the original color, and thus the blinking effect. Now for the fun. We would obviously like to do more that stare at a blinking cursor. At the least we should be able to move it around. That can be handled by a routine at line 300:

```
 300   OFF KBD:PERFORM xfroption(%0)
 305   PERFORM pencolor(%savecolor%):PERFORM dotrel(%0,%0)
 310   IF KBD>31 THEN GOSUB 360:GOTO 340
 320   IF KBD=27 THEN POP:GOTO 170
 325   xinc%=zip%*(( KBD=21)-( KBD=8)):yinc%=zip%*(( KBD=11)-( KBD=10))
 330   PERFORM moverel(%xinc%,%yinc%)
 340   savecolor%= EXFN%.xycolor:PERFORM xfroption(%4)
 345   ON KBD GOTO 300
 350   RETURN
 360   REM  commands go here
```

This one little routine does quite a bit. Line 300 and 305 just restore the state of the screen and turn off the keyboard interrupt to insure that the next statements are properly executed. Line 310 checks to see if the key pressed was a printable character; if so, line 360 starts a series of routines to process the command that the letter represents. If the character is a control character, it is checked for "Escape". Escape is used to signal quiting the screen and going back to main level options. Thus the POP statement is used to jump out without leaving our GOSUB hanging. If the character is not an escape, line 325 does some clever processing: through the use of logical statements, the character is checked for one of the arrow keys and the appropriate X or Y coordinate is incremented. Note that the variable "zip%" is used to multiply the effect of the increment, to enable large cursor moves. This is initialized in the subroutine at line 1000. If line 325 is confusing to you, take a moment to study its effect. Try out various values of KBD to see how it works. These logical statements (not available in many Basics) can replace a lot of clumsy and lengthy IF statements. Line 330 moves the cursor as appropriate, and the rest of the routine cleans up and returns to line 92. To wrap this routine up, and the one above it, we need to add a couple of lines:

```
 170   TEXT:GOTO 35
 180   PERFORM release:PERFORM release
 185   CLOSE:INVOKE
 190   END
```

Just a few more statements and we'll have a fully functional program! As was mentioned earlier, line 360 begins a subroutine that handles commands. We already have a way to move the cursor around on the screen, and the following lines implement the simple functions of "draw a dot", "erase a dot", speed up the cursor, and return the cursor movement to normal. We'll use the letters "D", "E", "Z" (for zip!) and "N" to describe those functions. The statements look like this:

```
 360   IF KBD>95 THEN key$=CHR$(KBD-32):ELSE key$=CHR$(KBD)
 366   IF key$="D" THEN PERFORM pencolor(%pen):PERFORM
dotrel(%0,%0):RETURN
```

```
 367   IF key$="E" THEN PERFORM pencolor(%fill):PERFORM
dotrel(%0,%0):RETURN
 368   IF key$="Z" THEN zip%=zip%*2:RETURN
 370   IF key$="N" THEN zip%=1:RETURN
 399   RETURN
```

The first line (360) just makes sure that lower case letters are upshifted, and then, for ease of reading mostly, the value is converted to an ASCII character. From there on, IF statements test the value and perform the functions. Notice that drawing and erasing is as simple as changing the pencolor and drawing a dot (relative plotting is used to save the trouble of getting the coordinates). The "Z" command just doubles the movement of the cursor each time it is pressed. This comes in handy, especially on the 560X192 screen. The RETURN on line 399 just traps any invalid commands which might be typed and returns with no effect. With this in mind, it is easy to add features:

```
 372   IF key$="H" THEN PERFORM moveto(%0,%0):RETURN
```

This just "homes" the cursor, in case you get it lost off the screen. Yes, thats right, you can move the cursor anywhere in that -32768 to 32767 space!

Now, since we already have a circle draw routine, we can add that very simply:

```
 374   IF key$="C" THEN INPUT r:GOSUB 450:GOSUB 900:GOSUB 460:RETURN
```

Notice we have added references to two new GOSUBs; 450 and 460. These are used to save and restore the state of the cursor, since the circle draw routine always leaves the cursor on the circle. They are simple, and look like this:

```
 450   xcen= EXFN%.xloc:ycen= EXFN%.yloc
 455   cres= EXFN%.xycolor:PERFORM pencolor(%pen):RETURN
 460   PERFORM moveto(%xcen,%ycen):PERFORM pencolor(%cres):RETURN
```

The other thing new about the routine at line 374 is that it asks for input.

The prompt will be displayed on the text screen, so you won't see it unless you chose to add a TEXT command to switch back before the INPUT. You would then need to PERFORM grafixon to get the screen back. Personally, I prefer it as shown.

Ok, this should be enough to make for an interesting display. Now would be a good time to type this program in and debug it (it is possible to make typing mistakes!).

Ok, now that you're back for more, try adding the following:

```
 376   IF key$="B" THEN INPUT w,h:GOSUB 450:GOSUB 500:GOSUB 460:RETURN
```

That's right, "B" stands for "Box" and uses a small subroutine at 500:

```
500   w=w*scalefac
```

```
510   PERFORM linerel(%w,%0):PERFORM linerel(%0,%h):PERFORM linerel(%(-
w),%0): PERFORM linerel(%0,%(-h))
```

Are you beginning to get the idea of how easy it is to add features to this package? You probably are starting to get some neat ideas of your own, but here are a few simple ones while you are thinking:

```
 380   IF key$="T" THEN SWAP pen,fill:PERFORM fillcolor(%fill):PERFORM
pencolor (%pen):RETURN
 382   IF key$="R" THEN xrem= EXFN%.xloc:yrem= EXFN%.yloc:RETURN
 384   IF key$="L" THEN GOSUB 450:PERFORM lineto(%xrem,%yrem):GOSUB
460:RETURN
 386   IF key$="X" THEN PERFORM fillcolor(%fill):PERFORM fillport:RETURN
```

Line 380 lets you "Toggle" beween fillcolor and pencolor. This is handy to erase something you just drew (like a circle or a box), by toggling the pencolor. Then you can repeat the previous command and it will magically disappear! Line 382 simply "remembers" a point. It is used in conjuction with line 384, which draws a line from that point to wherever the cursor is located. Line 384 creates the "X" command to completely erase the viewport. That's a good idea for a command you can add, to reset the current graphics viewport.

A couple more small ones and then we will get to the last two biggies. Here are two which permit you to save and load the graphics screen to disk. This is not only useful for making a permanent copy, but can be used before any big sequence of commands. In case you don't like the results, simple reload the old contents, and the screen is magically restored.

```
 388   IF key$="S" THEN PERFORM gsave."picture":RETURN
 390   IF key$="P" THEN PERFORM gload."picture":RETURN
```

If you are like me now, you've got so many commands in the program that you have to write them down. That's great. All huge programs were tiny subroutines once upon a time.

This next routine deserves some study. One of the things that would be nice, especially for charts and graphs, would be to put text on the graphics screen. Fortunately, The .GRAFIX permits that to be done easily. However, we can make this much more sophisticated with just a little programming effort, to wit:

```
392   IF key$="W" THEN GOSUB 450:GOSUB 600:GOSUB 460:RETURN
```

Subroutine 600 is used to "Write" on the screen. See how all these command letters make sense? - after a while the toughest part of the program is making up new commands which don't use any of the already taken letters! Anyway, here's the screen writing subroutine:

```
 600   charcnt=0:line$=""
 605   GET a$
 610   IF ASC(a$)<32 THEN 640
```

```
615   line$=line$+a$
620   PRINT#1;a$;
625   charcnt=charcnt+1
630   GOTO 605
640   chr=ASC(a$)
645   IF chr=13 THEN RETURN
650   IF chr<>8 OR (chr=8 AND charcnt=0) THEN 605
655   SWAP pen,fill
660   PERFORM pencolor(%pen)
665   PERFORM moverel(%-7,%0)
670   PRINT#1;RIGHT$(line$,1);
673   PERFORM moverel(%-7,%0)
675   charcnt=charcnt-1
678   line$=LEFT$(line$,charcnt)
680   SWAP pen,fill:PERFORM pencolor(%pen)
685   GOTO 605
```

The nice thing about this routine is that it not only writes on the screen as you type, but allows you to use the "back-arrow" to erase mistakes. To permit this, the variable "line$" keeps track of what you have typed, and if a back-arrow is encountered, the routine picks off the last character in the string, puts the pen in fill mode, and reprints the character on top of the original character, erasing it. Note that this backing up is done with the PERFORM moverel command, and -7 is used because that is a standard character space. Note also that you have to back up after erasing too, since the graphics routines still think you were writing an ordinary character.

A carriage return terminates the write mode, and a check in line 650 ensures that you are not allowed to back up past the beginning point.

The last routine is the most complex, and the one in the worst shape. By that is meant that it works fairly reasonably, but could stand enormous improvement. It was meant as a beginning, and any help, suggestions, modifications, etc. would be appreciated. The routine is the promised-from-last-time "area fill" subroutine. It is integrated into the package as follows:

```
394   IF key$="F" THEN GOSUB 450:GOSUB 1300:GOSUB 460:RETURN
```

The subroutine at line 1300 does the work, and looks like this:

```
1300   target=pen:startx%= EXFN%.xloc:starty%= EXFN%.yloc
1302   filled=0:inc=0:flag=0
1305   GOSUB 1400
1307   IF filled=1 THEN 1350
1310   PERFORM moveto(%startx%,%starty%)
1315   GOSUB 1430
1330   PERFORM linerel(%-(rxprev%-lxprev%),%0)
```

```
 1335   startx%=(rxprev%-lxprev%)/2+lxprev%+.5:starty%=starty%-1
 1340   PERFORM moveto(%startx%,%starty%)
 1345   GOTO 1305
 1350   IF inc=0 OR flag=1 THEN RETURN
 1351   flag=1
 1352   incval=inc
 1355   FOR ival=1 TO incval
 1360     startx%=(srch%(ival,2)-srch%(ival,1))/2+srch%(ival,1)+.5
 1365     starty%=srch%(ival,3):lxprev%=srch%(ival,1)
 1370     filled=0:PERFORM moveto(%startx%,%starty%):GOSUB 1305
 1375     NEXT ival
 1380   RETURN
 1400   IF EXFN%.xycolor<>target THEN 1410
 1401   IF flag=0 THEN
inc=inc+1:srch%(inc,1)=startx%:srch%(inc,2)=rxprev%:srch %
(inc,3)=starty%
 1402   IF startx%-lxprev%<=2 THEN filled=1:RETURN
 1404   startx%=(startx%-lxprev%)/2+lxprev%:PERFORM
moverel(%-(startx%-lxprev%),%0)
 1406   IF EXFN%.xycolor=target THEN 1402
 1410   FOR i=1 TO startx%
 1415     PERFORM moverel(%-1,%0):IF EXFN%.xycolor=target THEN lxprev%=
EXFN%.xloc:RETURN
 1417     NEXT i
 1420   lxprev%=0:RETURN
 1430   FOR i=startx% TO xdot(mode)
 1435     PERFORM moverel(%1,%0):IF EXFN%.xycolor=target THEN
rxprev%=i:RETURN
 1438     NEXT i
 1440   rxprev%=xdot(mode):RETURN
```

Messy, right? RIGHT! Unfortunately, there is no easy way to do general area fill. The principle is that you must first locate the cursor in the uppermost part of the figure to be filled. The routine then searches down and across for a match for its current pencolor. When found, the cursor returns to the starting point and searches right until a match is found. A line is then drawn from the right-hand point to the left. Subroutines at 1400 and 1430 do the left and right scanning.

Of course, if this was all it did, the routine would be a great deal simpler. The additional sophistication lies in an algorithm designed to enable the filling of complex shapes which contain other shapes. The most trival example is that of a circle within a circle. If you want to create something that looks like a donut, you could draw one circle inside the other, and fill the space between the two circles. This routine will handle most of those cases, along with circles inside boxes, etc. This is done by always favoring the right hand side of figures, and

putting information in the srch% array when the routine suspects that it may have missed something. In addition, the routine trys to begin searches from what it suspects is the center of the open area. The problems come when the figure inside a figure is sharply off to the right-hand side of the larger object. The routine will usually miss a part of the filling, because of an inadequate scan. Maybe by next time there will be a hotter version, but in the meantime, its easy to just move the cursor over and reissue the fill command to get what was missed.

After all that appology, it should be pointed out that any simple figure, expecially a convex one, will be filled reasonably well, as long as you start at the top. In fact, you might want to tweak this routine for simple figures by jettisoning the srch% array, and changing the search to look up as well as down. Note also that no real optimization was done. Using the usual tricks about multiple statements on a line, replacing constants, plus tighter coding would probably speed this routine up considerably. Oh well, another project for you in your spare time.

One last parting shot. Lots of times features are put in Basics which do not seem to be particularly useful. However, every command has some real purposes, and finding them can save lots of programming, and usually make your applications more efficient. In looking over the program above, there is a crying need for INSTR and ON GOSUB. Notice:

All of the "IF key$=" statements could be replaced with the following:

```
1057    command$="DEZNHCBTRLXSPWF"

362     cmd=INSTR(command$,key$)
364     ON cmd GOSUB
366,367,368,370,372,374,376,380,382,384,386,388,390,392,394
365     RETURN
```

Then each subroutine line could look like this example:

```
394   GOSUB 450:GOSUB 1300:GOSUB 460:RETURN
```

This is not only neater, but much more efficient, since it is not necessary to go through fifteen IF statements just to find the one that's wanted. It also permits multiple line subroutines, which the other structure does not. Adapting the other structure to multiple line routines would cause the IF statements to have to be linked with GOTO, a situation to be avoided.

Well, so much for philosophy. This has been a meaty article, hopefully. There are many features which need to be added to this month's package to make it truly useful, but by now surely you've dropped this magazine and are bent industriously over the keyboard. Time to tiptoe quietly away . . . Genius at Work!

Exploring Business Basic - Part IX

Department of Good Ideas

This section is called "Department of Good Ideas" to serve as a reminder that all the planning in the world about what Business Basic topics need to be covered can be undone by a simple question by a Basic user. This month's column is devoted to just such a simple question, asked by a programmer with a database application to implement: "How can I use random access files to look up records when the record numbers I want to use are non-numeric or exceed the 32767 record limit?"

We will return to graphics next month to explore character sets and animation, but for now, this question is fundamental to the requirements of lots of applications. In addition, this topic covers some interesting ground in computer science that everybody who writes interesting programs could benefit from.

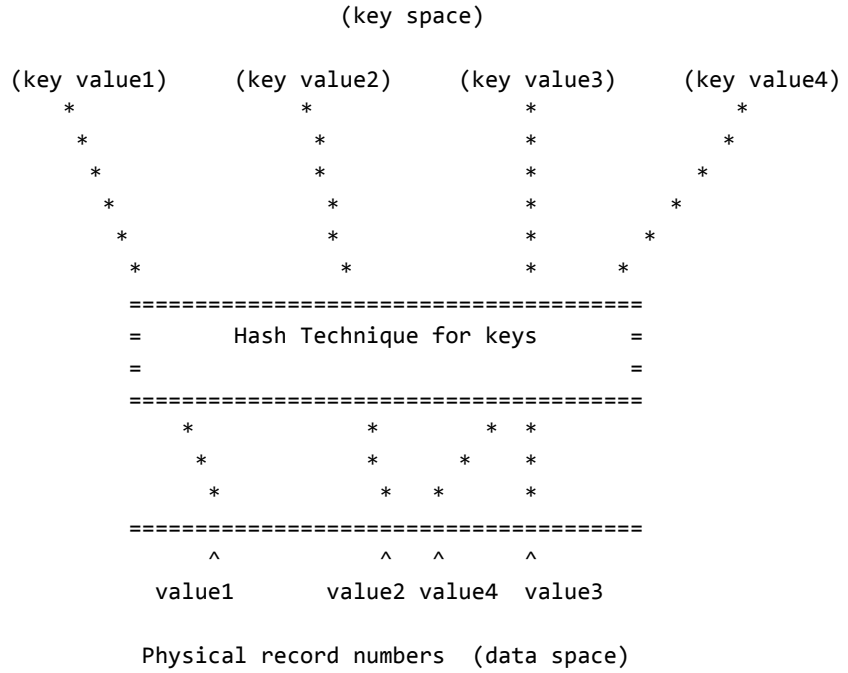
Slinging the Hash

The technique that our intrepid questioner needs to know about is something called "hashing". In general, this refers to using some mathematical operation on a value (string or numeric) to obtain a new value that is within the range of desired values. A typical example is the following: in a file which will maintain records on only 1000 employees, use their social security number (nine digits - 1,000,000,000 possible values) as a reference number for direct lookup of information. In this particular situation, a formula is required which will convert the nine digit number into a three digit number. The resulting three digit number then can be used to look up the employee record, assuming that the formula resolves each of the social security numbers into a unique three digit number.

It is in this area of resolving unique record numbers from more complex "key values" where hashing techniques get interesting. In our example, it is easy to imagine simply dropping the first six digits of the number to obtain a three digit result. In that case, "229-49-7128" becomes simply 128. In this way, "305-47-6024" would refer to 24, and "906-28-2935" would become record 935. Actually, in the case of Social security numbers, this technique is not all that bad. It is easy to see that there are many (a million to be exact) different social security numbers which end in a given three digits, but in a random selection of employees, the odds of many with the same last three digits are fairly small.

In a few minutes we'll see that the phrase "fairly small" will induce a significant amount of programming effort to deal with duplicates, but for now, consider that other "key" values (that is, values which are used as "keys" in looking up records) present even more interesting problems. Dealing with a key value like

"305-47-6024" may seem like a straightforward problem, but consider what "290-AR37BH" would do to our simple scheme of using the last three digits. In fact, there is no telling what the structure of many key values might be. Suppose that we used part numbers which all varied in the first three digits instead of the last! Each of our three digit "hashed" keys would be identical, thus rendering the whole scheme useless. A more ideal technique would be to perform operations on the entire key value which would generate a reasonably "random" value within the range of record numbers which our file could contain. If this generated value results in a "random" value, we can assume that the distribution of values in the "data space" (the set of all possible "hashed" record numbers) is reasonably uniform, with minimum conflicts. The diagram below represents the desired result:



There are many techniques for generating such hashed values from alphanumeric keys. The example program below will let you experiment with one such method:

```
10   PRINT"Hash key create program"
20   INPUT"Maximum record number: ";r$
22   IF r$="" THEN GOTO 50:ELSE:recordmax&=CONV&(r$)
25   INPUT"Your key value: ";a$
27   IF a$="" THEN 20
28   PRINT"Key length = ";LEN(a$)
30   GOSUB 1500
```

```
35   PRINT"Derived value is: "a&"  hash is: ";key&
40   GOTO 25
50   END
1500   a&=1:lstring=LEN(a$)
1502   FOR i=1 TO lstring
1505     ascval=ASC(MID$(a$,i,1))
1510     a&=a&+CONV&(ascval*2^i+ascval*3^(lstring-i+1))
1520     NEXT i
1525   key&=a& MOD recordmax&
1530   RETURN
```

This sample program first asks for your maximum record number. This creates a value, "recordmax&" which is used as the upper limit on hash key generation. The subroutine at line 1500 actually generates the hash value from the alphanumeric input in line 25. It works by going through each character position in the key and converting it to its ASCII equivalent (line 1505). Then a number is generated in line 1510 (ascval*2^i+ascval*3^(lstring-i+1)) by multiplying the value by a power of two dependent on the character position in the string, and adding the product of the value times a power of three equivalent to its position relative to the end of the string. This effectively generates considerably different numbers, even if the original value differed only by one in the last position. It also mimimizes duplicates resulting in reversing the order of the characters, which a simple sum would not. Once this calculated value is produced, it is reduced to the range of the "data space" by the modulus function MOD in line 1525. Remember that MOD gives the remainder of dividing by the "recordmax&" value, and thus guarantees a value between 0 and recordmax&-1. Line 35 prints the result of this calculation, so you can get a feeling for how different hash values are for some very similar key values. Type this program in and try it for various key values to be sure you understand what's going on.

Once you have tried this program with various key values, try rerunning it with a very small data space. In other words, use something like 11 or 7 for the maximum record number. You will quickly discover that lots of very different key values will produce the same hash value. This is the fundamental problem with hashing techniques, since each duplicate hash value represents a potential conflict in the file. Ah well, nothing good comes easy!

One way to test the ideal data space sizes against various numbers of records to hash is to use a file of random key values. The following program will create a "junkfile" filled with nine character key values for test purposes:

```
10   PRINT"Random key generation program"
15   OPEN#1,"junkfile"
25   INPUT"Number of records to generate: ";n
30   FOR ival=1 TO n
35     a$="         "
```

```
 40     FOR j=1 TO 5:SUB$(a$,j,1)=CHR$(65+INT(26*RND(1))):NEXT j
 45     FOR k=6 TO 9:SUB$(a$,k,1)=CHR$(48+INT(10*RND(1))):NEXT k
 47     PRINT a$
 60     NEXT ival
 80   CLOSE
 90   END
```

The only thing of real note in this program is the use of the SUB$ function to speed up the string generation, compared to the use of the "+" (concatenation) operator. In any case, this program will generate a file of random keys, with the first five positions alphabetic, and the last four numeric.

The program below will read this file and allow you to experiment with the size of the data space compared with the number of records to be loaded, and print out a simple picture of the number of conflicts.

```
 10   PRINT"Hash key evaluation program"
 12   PRINT:INPUT"Number of records in trial data space: ";rec
 15   recordmax&=CONV&(rec)
 20   DIM fill%(1000)
 22   OPEN#1,"junkfile"
 25   INPUT"number of records to read: ";n
 30   FOR ival=1 TO n
 35     INPUT#1;a$
 50     GOSUB 1500
 52     key=CONV(key&)
 55     fill%(key)=fill%(key)+1
 60     NEXT ival
 65   FOR i=0 TO CONV(recordmax&-1)
 70     PRINT fill%(i);
 75     NEXT i
 85   END
1500   a&=1:lstring=LEN(a$)
1502   FOR i=1 TO lstring
1505     ascval=ASC(MID$(a$,i,1))
1510     a&=a&+CONV&(ascval*2^i+ascval*3^(lstring-i+1))
1520     NEXT i
1525   key&=a& MOD recordmax&
1530   RETURN
```

A typical run of the program will produce output similar to the following:

```
Hash key evaluation program

Number of records in trial data space: 100
number of records to read: 50


020000600000000010000300001000040000300002000010000400002000020000200003
000020004000030000100004000
```

As can be seen from the program, each digit position of the printout represents a different key value, and the number in the position represents the number of key values which "hashed" to that location. Based on the output above, the hashing appears far from random. The conflicts bunch up at intervals of approximately five, with lots of empty space in between. You should get similar results, even with a different "junkfile". Now try the program again, with a slight change:

```
Hash key evaluation program

Number of records in trial data space: 97
number of records to read: 50

110220000000001011001000000000101001101010001020000000010001110000210121
100110000101012300220210 2
```

Notice that this time the entries are not nearly so regular. There are still conflicts (indicated by the "2"s and "3"s in the list), but they are scattered about without a definite pattern. You might argue that this distribution is not random, since there is still bunching up of values. Examine the following program, which does produce a reasonably random distribution, and see what happens:

```
 10   PRINT"Random distribution program"
 12   PRINT:INPUT"Number of records in data space: ";recordmax
 20   DIM fill%(1000)
 25   INPUT"Number of random numbers to generate: ";n
 30   FOR ival=1 TO n
 50     key=INT(recordmax*RND(1))
 55     fill%(key)=fill%(key)+1
 60     NEXT ival
 65   FOR i=0 TO recordmax-1
 70     PRINT fill%(i);
 75     NEXT i
 85   END
```

The result from your runs should look something like this:

```
Random distribution program

Number of records in data space: 97
Number of random numbers to generate: 50

00100200010111100100100000010001110111001210101100013002010000010000003000101100010102102121000001
```

Each time you run this program, the results will be different, but similar. True random distributions tend to be bunchy, and definitely non-uniform in the sense that there will typically be conflicts, unless the data space is very large in comparison to the number of entries.

Now comes the real question. If you were following along, you may have tried the last program with the first set of numbers (data space=100, entries=50). Notice that the same regular bunching occurs as occurred in the sample run with "junkfile". This suggests (although the actual proof is something we won't cover here) that more regular distributions can be obtained by using numbers like 97 instead of 100. Yes, there's only a difference of 3 between them, but in fact there is a much more important difference: 97 is a prime number, while 100 is obviously not. Using non-prime numbers as data space values is almost certain to create non-random bunching of record numbers, and thus lots of conflicts. The following simple program will rapidly allow you to pick prime numbers as candidates for data space values in your programs:

```
 5   INPUT"Range of prime number search: ";y,z
 10   IF z=0 THEN 80
 15   FOR j=y TO z
 20     IF j/2=INT(j/2) THEN 65
 30     FOR i=3 TO SQR(j) STEP 2
 40       IF j/i=INT(j/i) THEN 65
 50       NEXT i
 60     PRINT"The number "j" is prime"
 65     NEXT j
 70   GOTO 5
 80   END
```

If necessary, this program can easily be converted into a subroutine for use in larger programs which need to set data space sizes based on estimates on the total number of expected records in the file.

Summing up

The enormous volume of expository material above was designed to show ways to produce a random record number from an arbitrary collection of characters called a "key value". In the process we discussed the potential problem of conflicts, where two (or more) different key values would "hash" to the same record number. Dealing with these conflicts is the most challenging part of programming file access methods using hashing. Before we get into an actual database program which uses these techniques, it would be worthwhile to think about ways to reduce conflicts and improve performance.

Hash rule number 1

use a "hash" method which obtains as random as possible a distribution of physical record numbers. Remember that we used prime numbers as divisors in the examples above, in addition to doing a substantial amount of arithmetic on the key values themselves. There are other methods (any good reference will talk about "radix transforms", etc.) but the prime divisor method is a good all around choice.

Hash rule number 2

use as large a data space as possible, compared with the total number of expected records, so that the "hashed" records are spread out with minimum conflict.

On the Apple III we are fortunate to have a file system which allocates disk blocks only when they are used. This suggests that the actual "cost" of using large data spaces is not very significant. It is easy to imagine using a data space of approximately 5000 records to contain a probable maximum of 1000 physical records, since the actual overhead of such a scheme may only be a few extra index blocks. This kind of five to one ratio of data space to physical records will cut conflicts to the point where they do not impact performance. Compare this to randomizing 1000 records into a 1200 record space, where nearly every hashed record will conflict with another, and the probabilities are that some may have as many as four or five conflicts.

Hash rule number 3

For maximum performance, use the extra memory of the Apple III to maintain all conflict tables, and minimize the amount of shuffling of disk records required to resolve conflicts.

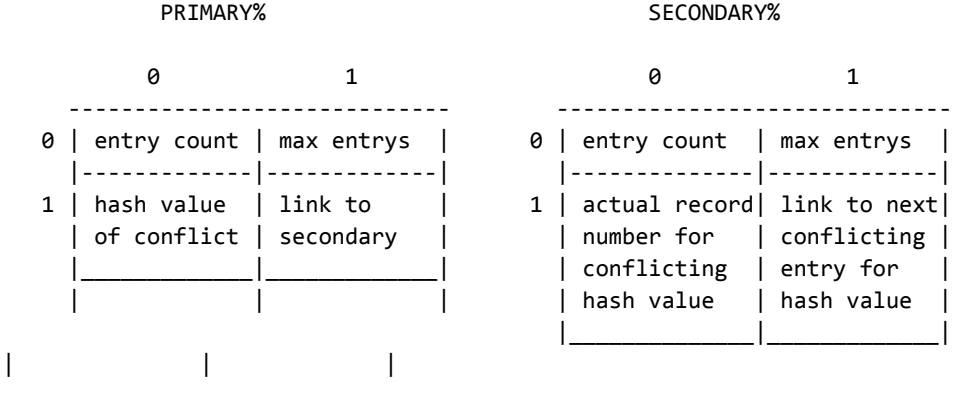
This rule seems like common sense, but remember that most hash techniques were developed in the mainframe computer days, when disks were fast and memory was expensive. Today's personal computer world is exactly the opposite, and requires a restructuring of the approach to "hashed" file access.

A Real Program

So far every thing which has been discussed has been theoretical. Hopefully you have done the exercises so that the following rather complex program can be absorbed in bite-sized chunks. For the application itself, "return with us now to those thrilling days of yesteryear", that is, the October and November columns, where we discussed a simple parts file application program which used four values: part number, description, location and quantity. Observe the following:

```
 5   DIM primary%(200,1),secondary%(300,1),trial%(100),chron%(1000)
 10   GOSUB 1980
 15   PRINT"Database program using HASH"
 20   PRINT:INPUT"File name: ";file$
 22   IF file$="" THEN 200
 25   OPEN#1,file$,45
 30   GOSUB 2100
```

In line 5 several arrays are set up to deal with pointer mechanisms which will be used later. "Primary%" contains the list of all records which are in conflict with other records previously entered. "Secondary%" contains the physical record numbers where these conflicting records are stored, along with a link to any other conflicting records which hash to the same value. "Trial%" is used later to maintain conflict lists for search purposes, and "chron%" contains a chronological list of all physical record numbers which have been used. The structure of "primary%" and "secondary%" are as follows:



The subroutine at line 1980 sets up these initial values and establishes a function (nospace) which checks to see if there is room left in the conflict lists for entrys:

```
 1980   modify=0:recordmax&=4951:maxprimary%=200:maxsecondary%=300
 1982   DEF FN
nospace(x)=(primary%(0,0)=primary%(0,1))+(secondary%(0,0)=second
       ary%(0,1))
 1990   RETURN
```

After requesting the database file name, the subroutine at line 2100 checks to see if the database file is already initialized, and if so, reads the contents of the conflict and chronological arrays into memory:

```
 2100   ON ERR GOTO 2150
 2105   datatype=TYP(1):IF datatype<>2 THEN 2150
 2110   READ#1,0;totprimary%:IF TYP(1)<>2 THEN 2150
 2112   READ#1;totsecondary%
 2113   ON ERR errorcode=2:GOTO 2140
 2115   FOR i=0 TO totprimary%
 2120     READ#1;primary%(i,0),primary%(i,1)
 2122     NEXT i
 2123   ON ERR errorcode=3:GOTO 2140
 2125   FOR i=0 TO totsecondary%
 2130     READ#1;secondary%(i,0),secondary%(i,1)
 2132     NEXT i
 2133   READ#1;chron%(0)
 2134   IF chron%(0)=0 THEN 2140
 2135   FOR i=1 TO chron%(0)
 2136     READ#1;chron%(i)
 2137     NEXT i
 2138   errorcode=0
 2140   OFF ERR:RETURN
 2150   errorcode=1:OFF ERR:RETURN
```

The variable "errorcode" is used extensively in this program to pass problem information back to the calling part of the main program. In this case, errors are flagged if the beginning of the file does not contain the proper data. Lines 35 through 60 determine if the database is initialized, and if not, take the proper course of action.

```
 35   IF errorcode=0 THEN 100
 40   IF errorcode<>1 THEN PRINT"The database is damaged.
Errorcode=";errorcode:STOP
 45   PRINT"The file ";file$;" is not a database file."
 47   IF datatype<>0 THEN 20
```

```
50   INPUT"Would you like to make it a database file? ";reply$
55   IF reply$<>"Y" AND reply$<>"y" THEN CLOSE:DELETE file$:GOTO 20
60   GOSUB 2000
```

If the database is to be created from scratch, the subroutine at line 2000 takes care of the initialization of all arrays and values, and then physically writes them to the newly created file.

```
2000   primary%(0,0)=0:primary%(0,1)=maxprimary%
2010   secondary%(0,0)=0:secondary%(0,1)=maxsecondary%
2015   chron%(0)=0
2017   WRITE#1,CONV(recordmax&)+120;0
2020   WRITE#1,0;primary%(0,1),secondary%(0,1)
2025   FOR i=0 TO primary%(0,1)
2030     WRITE#1;primary%(i,0),primary%(i,1)
2035     NEXT i
2040   FOR i=0 TO secondary%(0,1)
2042     WRITE#1;secondary%(i,0),secondary%(i,1)
2045     NEXT i
2050   FOR i=0 TO chron%(0)
2055     WRITE#1;chron%(i)
2060     NEXT i
2075   RETURN
```

Note: those of you who followed the article on "REQUEST.INV" a few months ago know of a faster way of doing file reads and writes. In larger implementations of this technique, these high performance options really come in handy.

After initialization of the internal variables, an option list is presented, and each of the options (add, delete, find and list) uses its own subroutine for the particular task:

```
100   PRINT"Type:"
105   PRINT"     1 to add a record"
110   PRINT"     2 to delete a record"
115   PRINT"     3 to find a record"
120   PRINT"     4 to list all records"
155   PRINT:INPUT"Your selection: ";a$
160   IF a$="" THEN 200:ELSE:a=CONV(a$)
162   IF a<1 OR a>4 THEN 170
165   ON a GOSUB 500,600,700,900
170   PRINT:GOTO 100
```

Let's look at first things first, examining the "add" routine in the subroutine at line 500:

```
500   PRINT:INPUT"Part number: ";part$
505   IF part$="" THEN RETURN
```

```
 510   IF LEN(part$)>10 THEN PRINT"Part number too long, reenter":GOTO
500
 520   PRINT:INPUT"Description: ";desc$
 530   IF LEN(desc$)>15 THEN PRINT"Description too long, reenter":GOTO
520
 535   PRINT:INPUT"Location: ";loc$
 540   IF LEN(loc$)>10 THEN PRINT"Location too long, reenter":GOTO 535
 545   PRINT:INPUT"Quantity: ";quan$
 550   q=CONV(quan$):IF q>9999 THEN PRINT"quantity too large,
reenter":GOTO
545
 555   quantity%=q
 560   PRINT:PRINT"Record is: "part$"|"desc$"|"loc$"|"quantity%"|, ok?
";
 565   INPUT"";a$
 570   IF a$<>"Y" AND a$<>"y" THEN PRINT:GOTO 500
```

This part is pretty straightforward. It simply accepts the values, does minimal editing for length and value, and then reprints the record in line 560 to allow the user to verify that everything was correctly entered.

Next, things get a bit sticky.

```
 575   a$=part$
 580   GOSUB 1500
```

Line 1500 contains our old familiar routine, hashing a record number from the "part number" value:

```
 1500   a&=1:lstring=LEN(a$)
 1502   FOR i=1 TO lstring
 1505     ascval=ASC(MID$(a$,i,1))
 1510     a&=a&+CONV&(ascval*2^i+ascval*3^(lstring-i+1))
 1520     NEXT i
 1525   key&=a& MOD recordmax&
 1530   RETURN
```

The next sequence of events adds 100 to the resulting record number, to clear all the data we might want to write to the beginning of the file, and then calls the routine at line 1800 to actually determine and deal with the writing of the record, and the conflicts, if any occur:

```
 585   recordnum%=CONV%(key&)+100
 590   GOSUB 1800
 595   IF errorcode=1 THEN PRINT"Tables full, cannot add a conflicting
record."
       :RETURN
 597   PRINT:PRINT"record added.":RETURN
```

The "add" routine at 1800 is non-trivial. It first determines (in line 1800 to line 1810) if a conflicting record already exists in the "primary%" conflict list. Line 1807 is particularly interesting in that, as we shall see later in the "delete" routine, flagging a conflict with a negative sign means that the conflicting record has been deleted and can be reused.

Note that after scanning the table, line 1815 and 1820 check to see if the physical record contains a string value as its first variable. If not, the record is considered available. If not, the record is considered occupied, with the initial string variable equal to the "part number", which is actually the element we use as the hash key.

```
 1800   FOR i=1 TO primary%(0,0)
 1805     IF primary%(i,0)=recordnum% THEN 1830
 1807     IF ABS(primary%(i,0))=recordnum% THEN primary%
(i,0)=recordnum%:GOTO 1815
 1810     NEXT i
 1815   READ#1,recordnum%
 1820   IF TYP(1)<>4 THEN 1900
 1821   trialrec%=recordnum%+1:lookup%=0
 1822   IF FN nospace(x) THEN errorcode=1:RETURN
 1823   primary%(0,0)=primary%(0,0)+1:currentp%=primary%(0,0)
 1824   primary%(currentp%,0)=recordnum%
 1825   secondary%(0,0)=secondary%(0,0)+1:currents%=secondary%(0,0)
 1826   primary%(currentp%,1)=currents%
 1829   GOTO 1855
```

Lines 1821 though 1929 deal with first-time conflicts, and create a new primary record along with locating a place to enter the physical secondary record number. This record number is obtained by the routine starting at line 1855. The routine at 1855 is also used in the event that the normal scan of primary conflict records (line 1805 above) discovers a duplicate entry. The routine at line 1830 searches the list of primary and secondary records until the end of the conflict list is found. At that point "trialrec%" is set to the next suspected available record, and execution goes to 1855 to find a physical record into which to put our entry. Note that line 1837 ensures that deleted entrys in the conflict list are automatically reused.

```
 1830   IF FN nospace(x) THEN errorcode=1:RETURN
 1835   lookup%=primary%(i,1)
 1837   IF secondary%(lookup%,0)<0 THEN
recordnum%=ABS(secondary%(lookup%,0)):s
        econdary%(lookup%,0)=recordnum%:GOTO 1900
 1840   link%=secondary%(lookup%,1)
 1845   IF link%<>0 THEN lookup%=link%:GOTO 1837
 1847   secondary%(0,0)=secondary%(0,0)+1:currents%=secondary%(0,0)
```

```
 1850   trialrec%=secondary%(lookup%,0)+1
 1855   READ#1,trialrec%
 1860   IF TYP(1)<>5 AND TYP(1)<>1 THEN trialrec%=trialrec%+1:GOTO 1855
 1865   IF lookup%=0 THEN 1880
 1870   secondary%(lookup%,1)=currents%
 1880   secondary%(currents%,0)=trialrec%
 1885   secondary%(currents%,1)=0
 1890   recordnum%=trialrec%
```

Note that lines 1865 through 1890 add the new conflict list entry to the list in "secondary%" and set the physical record number "recordnum%" to the final trial record value.

```
 1900   WRITE#1,recordnum%;part$,desc$,loc$,quantity%
 1902   chron%(0)=chron%(0)+1:chron%(chron%(0))=recordnum%
 1905   errorcode=0:modify=1:RETURN
```

Line 1900 through 1905 then actually write the record values to the file, add the record number to the chronological list, and set the "modify" flag to let the program know that a change has been made to the file and the arrays.

Notice also that the path through all this code is extremely trivial if there is no conflict in the use of record numbers. In that case, execution sails through the loop in lines 1800-1810, checks the record for previous contents in lines 1815 and 1820, and finding none, jumps to line 1900 to write the record and update the list. As long as there are no conflicts, this technique is very fast, and even with conflicts, there is a minimum of searching for a free record as long as the data space is significantly larger than the total number of records.

Having covered using hashing to add records, finding records becomes somewhat the reverse process of going back through the lists:

```
 700   PRINT:INPUT"Part number: ";part$
 705   IF part$="" THEN RETURN
 710   IF LEN(part$)>10 THEN PRINT"Part number too long, reenter":GOTO
700
 712   a$=part$
 715   GOSUB 1500
 720   recordnum%=CONV%(key&)+100
 725   GOSUB 1600
 730   IF errorcode=1 THEN PRINT"Part number not found.":GOTO 700
 735   PRINT:PRINT"Part number: ";part$
 740   PRINT"Description: ";desc$
 745   PRINT"Location:    ";loc$
 750   PRINT"Quantity:    ";quantity%
 755   PRINT:INPUT"Press return to continue: ";a$
 760   RETURN
```

After collecting the part number and generating the hase value using the subroutine at 1500, line 725 goes to a subroutine which looks up records in the database. The tricky part about this is that there may be multiple records which have the save hash key (that is, are in conflict), so that is is necessary to assemble a list of all values from the primary and secondary conflict arrays, and then lines 1671-1692 read each record to determine which one is the actual one being sought. Note also that there is code in lines 1607 and 1662 to deal with the deleted entrys in the conflict lists.

```
 1600   FOR i=1 TO primary%(0,0)
 1605     IF primary%(i,0)=recordnum% THEN 1640
 1607     IF primary%(i,0)<>ABS(recordnum%) THEN 1610
 1608     listnum%=0:trial%(0)=0:lookup%=primary%(i,1):GOTO 1670
 1610     NEXT i
 1615   READ#1,recordnum%
 1620   IF TYP(1)=4 THEN GOSUB 1690:ELSE errorcode=1:RETURN
 1622   conflict=0
 1625   IF part$=part1$ THEN errorcode=0:RETURN:ELSE errorcode=1:RETURN
 1640   listnum%=0:trial%(0)=0
 1642   trialrec%=recordnum%
 1645   lookup%=primary%(i,1)
 1650   listnum%=listnum%+1
 1655   trial%(listnum%)=trialrec%
 1657   trial%(0)=trial%(0)+1
 1660   IF lookup%=0 THEN 1670
 1662   IF secondary%(lookup%,0)>0 THEN trialrec%=secondary%
(lookup%,0):skip=0: ELSE:skip=1
 1665   lookup%=secondary%(lookup%,1)
 1667   IF skip THEN 1660 ELSE 1650
 1670   conflict=1
 1671   FOR i=1 TO trial%(0)
 1672     record%=trial%(i)
 1673     READ#1,record%
 1675     IF TYP(1)<>4 THEN 1680
 1676     GOSUB 1690
 1677     IF part1$=part$ THEN errorcode=0:RETURN
 1680     NEXT i
 1682   errorcode=1:RETURN
 1690   READ#1;part1$,desc$,loc$,quantity%
 1692   RETURN
```

The last big section of the program deals with deleting records, and while it has been alluded to above, it is being mentioned third in the sequence of functions simply because it uses the "find" routines to locate the record to be deleted.

```
 600   PRINT:INPUT"Part number: ";part$
```

```
 605   IF part$="" THEN RETURN
 610   IF LEN(part$)>10 THEN PRINT"Part number too long, reenter":GOTO
700
 612   a$=part$
 615   GOSUB 1500
 620   recordnum%=CONV%(key&)+100
 625   GOSUB 1600
 630   IF errorcode=1 THEN PRINT"Part number not found.":GOTO 600
 635   PRINT:PRINT"Delete: "part1$"|"desc$"|"loc$"|"quantity%"| ? ";
 637   INPUT"";a$
 640   IF a$<>"Y" AND a$<>"y" THEN PRINT"Not deleted":RETURN
```

The first part of "delete" simply takes the part number information, hashes the key and then in line 625, gosubs to the "find" routine to locate the particular part number record. If the record is found, the user is asked to confirm that it is the proper record to delete, and then the fun begins:

```
 645   IF conflict=1 THEN 660
 650   record%=recordnum%
 655   GOTO 690
```

"Conflict" is a flag set in the find routine which tells "delete" whether or not there is cleanup work to be done in the conflict lists. If not, the record number is passed to 690 for physical deletion. If there is a conflict, then 660-670 find the primary entry, check if that is the physical record number to be deleted. If so, line 675 negates the entry. If not, the secondary list is searched in line 680-688 until the proper entry is found and flagged. Then 690-695 physically deletes the record and finds the entry in the chronological list, negating that as well. Because entrys are being changed, the modify flag is set in 695.

```
 660   FOR i=1 TO primary%(0,0)
 665     IF primary%(i,0)=recordnum% THEN 675
 670     NEXT i
 672   PRINT"Error in delete. Record not found":RETURN
 675   IF primary%(i,0)=record% THEN primary%(i,0)=-record%:GOTO 690
 680   lookup%=primary%(i,1)
 682   IF secondary%(lookup%,0)=record% THEN secondary%(lookup%,0)=-
record%:GOTO 690
 685   IF secondary%(lookup%,1)=0 THEN 672
 687   lookup%=secondary%(lookup%,1)
 688   GOTO 682
 690   WRITE#1,record%;0
 692   FOR i=1 TO chron%(0)
 693     IF chron%(i)=record% THEN chron%(i)=-record%:GOTO 695
 694     NEXT i
 695   PRINT:PRINT"Record deleted":modify=1:RETURN
```

At Last, The End

The final routine is this program is the "list", which is the simplest of all:

```
900   IF chron%(0)=0 THEN PRINT"No records to list":GOTO 930
905   FOR i=1 TO chron%(0)
906     IF chron%(i)<0 THEN 920
907     READ#1,chron%(i)
908     IF TYP(1)<>4 THEN 920
910     READ#1;part$,desc$,loc$,quantity%
915     PRINT USING"10a,2x,15a,2x,10a,x,4#";part$,desc$,loc$,quantity%
920     NEXT i
930   PRINT:INPUT"Press RETURN to continue: ";a$
935   RETURN
```

"List" simply goes through the chronological array, reads the physical record numbers (skipping deleted entrys in line 906) and formats the information into a list. What a treat to see a simple, straightforward routine for once!

Final wrapup is all that is left:

```
200   PRINT:PRINT"end of program"
210   IF NOT modify THEN 220
211   count=0
212   FOR i=1 TO chron%(0)
214     IF chron%(i)>0 THEN count=count+1:chron%(count)=chron%(i)
216     NEXT i
218   chron%(0)=count
219   GOSUB 2020
220   CLOSE:INVOKE
230   END
```

These lines handle "quitting", checking the "modify" flag and writing out the data if necessary. Note that before writing out the data, a cleanup is done on the "chron%" list to remove deleted entrys.

Really The End

This has been a long and tough exercise, and you deserve a break. Go off to the refrigerator, get a cool beverage of your favorite persuasion, and consider the fact that the program above can be easily modified to maintain almost any kind of data records, and the list routine can be used in conjunction with sorts and calculations to format almost any kind of report. The fact that this kind of capability can be developed in Basic is a tribute to the power of the Apple III, Business Basic and SOS, and not a bad testimony on your investment. Just one word of caution is in order. For simplicity, many of the errorchecking

routines which would be needed to turn this into a real application have been left out. If you get serious about using these kind of techniques, take the time to anticipate all the things that could go wrong and put in tests for them. Also, to learn more about the techniques alluded to in this article, which fit the general category of "Access Methods", check your library for books on data structures, database theory and indexed access methods. And then have another cool one....

Exploring Business Basic - Part X

First things first. I must offer an apology for being absent from these pages last month. Sometimes the press of earthbound duties prevents the treading of etherial pathways. Ah, well, here's hoping that this issue will keep you busy enough to make up for the gap in our cycle of exploration.

On the subject of exploration, congratulations are due to John Jeppson for the excellent couple of articles on Basic and the Apple III. The April issue which covered a character set editor was a fine one. If you haven't seen it, check for a back issue copy, it was really that excellent.

On With It

This month's article could really be entitled "Down and Dirty in the SOS mines" or perhaps more accurately "The .CONSOLE driver is your friend". Last article, you will perhaps recall, we looked at techniques which substituted a little thinking about how to access records by key values for the brute force of having the computer scan hundreds of records looking for the one we wanted. The result was a method which was both fast and flexible. In general, there is no substitute for taking a little time to think about the best way to implement a program, before beginning to write it.

Along with choosing the best method to implement a particular task, a good programmer always looks at the tools at hand. Since every computer and every implementation of Basic is different, it pays to check out the computer on which the program will be running, and see what features and capablities it can lend to the task. Sure, some will insist that it is most important to write programs which can run on as many computers as possible, but the truth is that some modification is always required, so why not make the best of the environment you have (especially if that environment can save you some work).

Basic with "hot SOS"

The inspiration for this month's article came from a friend who was complaining that it was hard to write in assembly language on the Apple III and use that code in a Basic program. After letting him know about the technical note on writing Invokable modules, the question of what he wanted to do happened to come up. As expected, he wanted to do special handling of input for some menu screens and most interestingly, he also wanted to handle the listing of records longer than 80 columns to the screen, and was perplexed as to how to do it without having to rewrite the screen each time. After seeing how well Visicalc did its vertical and horizontal scrolling, he was even more convinced that assembly language was the only way to go.

Before this article is concluded, we'll see how to do these things and more using Basic and the power of SOS! We have previously done some work with the Basic invokable module called "request.inv", found on the Business Basic product disk. That article covered the two functions "filread" and "filwrite" which correspond to the SOS F_READ and F_WRITE calls. This month we will concentrate on "control" and "status" functions of the "request.inv" module, which correspond to the SOS D_CONTROL and D_STATUS calls. Remember that SOS views files as one of two fundamental types, "character" and "block". The control and status functions we will use this time are generally applicable to character files, and are documented in the respective manuals for the devices.

Since its already been announced that this issue will deal with the .CONSOLE driver (that wonderful driver that you communicate with via keyboard and screen), your next logical step is to get out your "Standard Device Drivers" manual, and prepare to follow along with the fascinating discussions to follow. The discussion of the .CONSOLE driver starts on page 27 of the manual, and our first task is to get familiar with the capabilities it provides.

Some further CONSOLEation

The console on the Apple III really consists of two devices, the keyboard (normally a read-only device) and the screen (normally a write-only device). To communicate with these devices, especially to change their operating characteristics, SOS allows you to give and receive information from the "driver", a software routine which is responsible for managing the physical screen and keyboard hardware. Information about the driver is obtained by using the "status" commands, and changes are made through the "control" commands. Let's start with "status".

The following program uses the "request.inv" invokable module to allow calls to determine the status of the console:

```
10   DIM statuslen(18)
15   DATA 8,41,2,1,1,1,6,-1,5,-1,1,1,1,1,1,1,2,1,-1
20   INVOKE"/basic/request.inv"
25   FOR i=0 TO 18:READ statuslen(i):NEXT
30   device$=".console"
```

These first lines do some initialization. Since the status call will always return the buffer string fully padded to 255 characters in lenght, the "statuslen" array is set to the number of valid bytes returned by each different call. Note that some calls are reserved, and therefore invalid, and are indicated by a -1 in the data list. Those of you who are comparing the data statement with the descriptions of the status calls in the Device Drivers manual will notice some anomilies in the list, but have faith, it will all be explained later. Note also that the INVOKE statement

needs to be changed to refer to the proper pathname for "request.inv" on your system. Meanwhile, on with the code:

```
 35   INPUT"Status code: ";stat$
 36   IF stat$="" THEN 100
 37   stat=CONV(stat$)
 38   IF stat<0 OR stat>18 THEN PRINT"Status code out of range, try
again":GOTO 35
 39   IF statuslen(stat)<0 THEN PRINT"Invalid status code, try
again":GOTO 35
 40   buffer$=""
 42   PERFORM status(%stat,@buffer$)device$
```

Lines 35 through 42 check the status request for validity, and then use the PERFORM statement to make the SOS call. Remember that "status" returns its result in the string variable "buffer$" and the "@" symbol instructs Basic to pass the address (location) of the buffer$ variable, so that "status" can return the requested information in the string. Now that buffer$ contains the mystery information, we can print it out in meaningful form with the following:

```
 45   endlist=statuslen(stat)
 47   line$=""
 50   FOR i=1 TO endlist
 55     hexvalue$=HEX$(ASC(MID$(buffer$,i,1)))
 57     char=TEN(hexvalue$):IF char<32 THEN char=char+128
 58     char$=CHR$(char)
 59     line$=char$+"  "
 60     PRINT USING"2a,x";MID$(hexvalue$,3,2);
 65     IF i/26=INT(i/26) THEN PRINT:PRINT line$:line$=""
 70     NEXT i
 72   PRINT:PRINT line$:PRINT
```

Note that "endlist" is the number of valid bytes of status information, so the routine from line 50 to 70 scans "buffer$" and prints out the HEX value of each character. The "line$" string accumulates the ASCII character values (after changing "real" control characters to printable control characters in line 57) so they can be printed out below the HEX values. Line 65 handles the case where more characters need to be printed than can fit on a single line and line 72 cleans up after the last line of HEX values is printed.

All that remains is to return to request the next status code, and to provide a place to go to terminate the program.

```
 80   GOTO 35
 100   INVOKE
 110   END
```

Lets look at some sample output for a few typical "status" calls:

First, something simple. According to the manual, status 0 should do nothing.

The request module does a little better than that, returning the name of the device being accessed, to wit:

```
2E 63 6F 6E 73 6F 6C 65
```

More interesting, and twice as mysterious is status 1, which returns the total state of the console driver. For reasons of compatibility with printers, only the printable ASCII characters are listed below. All others can be deduced by their HEX equivalents above. As you fool around with other status calls, the order in which these status indications fall will become apparent. In the meantime, here's everything you wanted to know about the console, and then some!

```
28 00 01 0B 67 F3 01 01 0E 67 F3 03 00 80 0D 00 80 80 80 80 80 82 0D 00
17 00
(           g  s           g  s

4F 00 17 0F 00 82 0D 00 17 00 4F 00 17 0F 00
O                             O
```

Note that on your screen, the special character symbols will actually print out.

Now for something simpler, and more useful. Status code 2 is advertised as indicating the status of the "line termination character." Of course, you know the line termination character as the RETURN key or perhaps you have noticed that RETURN and CTRL-M are the same character. Notice the two bytes returned from a status 2 call:

```
80 0D
```

The first byte indicates (as described in the Device Drivers manual) that a specific line termination character is enabled. The second byte ("0D") is hex notation for decimal 13, or the CTRL-M (RETURN) character. Later on we'll see that this can be changed to any other character, for fun and profit.

Now come two calls which are interesting, but fraught with danger if you fool with them. Specifically they are status 6 and 8, the "attention" and "any key" events. Events can be set in SOS to trigger interrupts to interpreters (like Basic) to inform the interpreter that some specific action is required or requested. Again, as described in the Drivers manual, status 6 gives the priority and event ID of the attention event, along with the address of the event handler which services the event, and the character code which triggers the event. The return from status 6 should look something like this:

```
01 01 0E 67 F3 03
         g  s
```

Again, only the printable ASCII characters are listed. The rest will show up on your screen when you run the program. Note that the first two bytes are the

priority and the event ID, and the next three bytes compose the address of the event handler for this event. For those of you who wondered how Apple III addresses all that memory, part of the answer is here. Three byte addresses are used throughout most of the system code to allow addressing up to a theoretical limit of 512K bytes. The important thing to remember is that you should not change the event handler addresses, since that is an easy way to cause the system to leap off into space, never to return. The real key value in this call is the last byte, the attention character itself. As you can see, it is HEX 03, which is also ASCII character 3, usually known as CTRL-C. Yes, all those times you pressed CTRL-C to stop a program or break a listing, you were setting an "event" within the console driver, which flagged SOS to call Basic's event handler to shut down the current activity. Later on we will discuss how to change the attention character to something besides CTRL-C, which is guaranteed to baffle your friends. For now, consider the "Any key" event, number 8 on your hit parade:

```
00 01 0B 67 F3
         g  s
```

Same format, except with a different priority and different event handler address. In addition, there is no event character, since this event is set by the pressing of "any key". As you might expect, this is the mechanism which Basic uses to implement the ON KBD statement.

Other status calls of interest include the one for "cancel status" (yes Virginia, you can avoid CTRL-X printing the backslash character and skipping to the next line) and "backspace status" which determines if the backarrow is destructive (erases the character backed over) or non-destructive (so you can use the forward arrow copy feature). All this and more is yours to investigate with the status call.

Still Curious?

You now have a program which will do all the status calls listed in the Standard Device Drivers manual. Ah, yes, you say: "What about status call 18, PRESERVE VIEWPORT?" The truth is, there is a little problem with the way that this invokable is written. Status call 18 (for those who are not following along in the manual) saves the entire contents of the viewport into the buffer. Unfortunately, individual Basic strings are limited to 255 characters, and any attempt to use a string array in this invokable will illicit a polite error message. Worse yet, an attempt to save a viewport of more than 255 characters using the status call will lock up Basic as the invokable module happily writes data into the middle of Basic itself. For that reason, the "statuslen" array doesn't allow the use of that call. For those of you who are more adventurous, or who use viewports of less that 256 characters, consider the following:

```
 41   IF stat=18 THEN GOSUB 125
 43   IF stat=18 THEN GOSUB 140

 125   vtemp= VPOS:htemp= HPOS
 130   PRINT
CHR$(26);CHR$(0);CHR$(0);CHR$(2);CHR$(26);CHR$(9);CHR$(24);CHR$(3)
       ;CHR$(12);
 135   RETURN
 140   TEXT:VPOS=vtemp:HPOS=htemp
 145   RETURN
 200   DATA 8,41,2,1,1,1,6,-1,5,-1,1,1,1,1,1,1,2,1,243
```

These changes to the program will make a special case of the "save viewport" call. Basically it works like this: line 125 saves the current cursor position, and line 130 sets up a window consisting of the first three lines of the screen. Similar action could be taken using the Basic WINDOW command, but it doesn't hurt to see how console commands are use to accomplish the same purpose. In any case, setting up the window (which contains 240 characters) gives the status function something to read within the 255 character limit of the buffer$ variable. Line 140 restores the cursor to its original position, and since we changed line 200 to indicate 243 characters expected (240 + 3 bytes of coded information) the program will list out the screen data as it does the other types of status information.

To give you an idea of how this works, imagine that the following three lines are at the top of the display when you run the program above with status function 18:

```
 TYPE   BLKS  NAME            MODIFIED TIME  CREATED  TIME   EOF
 TEXT   00003 STATUS.LIST     05/23/82 19:31 05/23/82 19:31  933
 TEXT   00004 TRY.LIST        05/23/82 20:16 05/23/82 20:16  1189
```

The resultant listing of the buffer (made by lines 50 through 72) will look something like this:

```
82 4F 02 A0 D4 D0 A0 A0 CC D3 A0 C1 C5 A0 A0 A0 A0 A0 A0 CF C9 C9 C4 D4
CD A0
   O        T  P        L  S     A  E                    O  I  I  D  T
M

C3 C5 D4 C4 A0 C9 C5 A0 C5 C6 A0 A0 A0 A0 A0 A0 A0 A0 D9 C5 A0 C2 CB A0
CE CD
C  E  T  D     I  E     E  F                          Y  E     B  K
N  M

A0 A0 A0 A0 A0 A0 CD C4 C6 C5 A0 C9 C5 A0 D2 C1 C5 A0 D4 CD A0 A0 CF A0
A0 A0
                  M  D  F  E     I  E     R  A  E     T  M        O
```

```
A0 A0 A0 A0 A0 A0 D4 D8 A0 A0 B0 B0 A0 D4 D4 D3 CC D3 A0 A0 A0 B5 B2 AF
B2 B1
                  T  X        0  0     T  T  S  L  S           5  2  /
2  1

BA B1 B0 AF B3 B8 A0 B9 B3 A0 B9 B3 A0 A0 A0 A0 A0 A0 A0 A0 C5 D4 A0 B0
B0 B3
:  1  0  /  3  8     9  3     9  3                          E  T     0
0  3

D3 C1 D5 AE C9 D4 A0 A0 B0 AF B3 B8 A0 B9 B3 A0 B5 B2 AF B2 B1 BA B1 A0
B3 A0
S  A  U  .  I  T        0  /  3  8     9  3     5  2  /  2  1  :  1
3

A0 A0 A0 A0 A0 A0 A0 A0 D4 D8 A0 A0 B0 B0 A0 D2 AE C9 D4 A0 A0 A0 A0 B5
B2 AF
                        T  X        0  0     R  .  I  T              5
2  /

B2 B2 BA B6 B0 AF B3 B8 A0 B0 B1 A0 B1 B8 A0 A0 A0 A0 A0 A0 A0 A0 C5 D4
A0 B0
2  2  :  6  0  /  3  8     0  1     1  8                          E  T
0

B0 B4 D4 D9 CC D3 A0 A0 A0 A0 B0 AF B3 B8 A0 B0 B1 A0 B5 B2 AF B2 B2 BA
B6 A0
0  4  T  Y  L  S              0  /  3  8     0  1     5  2  /  2  2  :
6

B1 B9 A0 A0 A0 A0 A0 A0 A0
1  9
```

Just a bunch of gibberish, right? Unfortunately, computers are eminently logical, and our favorite, the Apple III, is no exception. Because of the way the video is organized for accesses over its internal 16 bit bus (which also provides transparent access to extended address bytes), the characters are mapped in an alternating pattern, with visually ajacent bytes split by a distance of half the viewport window. All that leads to the following adjustment of our program, to reconstruct the image from the scramble in the buffer:

```
 75   IF stat=18 THEN GOSUB 160:GOSUB 140

 160   INPUT"Press RETURN to reconstruct the captured display: ";a$
 165   buf1$=MID$(buffer$,4,240)
```

```
 170   HOME:PRINT CHR$(21);"5";
 172   FOR j=1 TO 3:VPOS=j:HPOS=1:FOR i=1 TO 40:PRINT
MID$(buf1$,80*(j-1)+i,1);
       MID$(buf1$,80*(j-1)+i+40,1);:NEXT i:NEXT j
 175   TEXT:RETURN
```

You can adjust the constants in line 72 to accomodate other viewport sizes. Note also that in line 170 the statement PRINT CHR$(21);"5"; is used to turn off several console options (like scrolling and new line) to insure the data is written back correctly. The TEXT statement in 175 sets everything back to normal.

Getting Control

Now that you have all this information, the immediate reaction is "how do I change things?" The following is a sample program which allow the use of the "control" call to modify the state of the console driver to your wishes. Remember, modifying certain things (like event handler addresses) can cause the system to crash, so try to keep it simple. With that warning, here's the magic incantation:

```
 10   DIM controllen(18)
 15   DATA -1,40,2,1,1,0,6,-1,5,-1,1,1,1,1,1,1,-1,73,-1
 20   INVOKE"/basic/request.inv"
 25   FOR i=0 TO 18:READ controllen(i):NEXT
 30   device$=".console"
 35   INPUT"Control code: ";ctrl$
 55   errorcode=0
 60   IF ctrl$="" THEN 145
 65   ctrl=CONV(ctrl$)
 70   IF ctrl<0 OR ctrl>18 THEN PRINT"Control code out of range, try
again":GOTO 35
 75   IF controllen(ctrl)<0 THEN PRINT"Invalid control code, try
again":GOTO
35
 80   ON ctrl+1 GOSUB
1000,1100,1200,1300,1400,1500,1600,1700,1800,1900,2000,2100,2200,2300,24
00,2500,2600,2700,2800
 85   IF errorcode THEN PRINT"Control function not performed.":GOTO 35
 90   PERFORM control(%ctrl,@buffer$)device$
 95   endlist=controllen(ctrl)
 97   line$=""
 100   FOR i=1 TO endlist
 105     hexvalue$=HEX$(ASC(MID$(buffer$,i,1)))
 110     char=TEN(hexvalue$):IF char<32 THEN char=char+128
 115     char$=CHR$(char)
 117     line$=char$+"  "
```

```
 120     PRINT USING"2a,x";MID$(hexvalue$,3,2);
 125     IF i/26=INT(i/26) THEN PRINT:PRINT line$:line$=""
 130     NEXT i
 135   PRINT:PRINT line$:PRINT
 140   GOTO 35
 145   INVOKE
 150   END
```

As you can see, the main part of the program is somewhat similar to our "status" example above. The big exception is the ON GOSUB statement in line 80 which allows for each control request to be handled separately. Here are the routines and commentary:

```
 1000   buffer$=CHR$(0)
 1010   RETURN
```

This is just the "reset console" function which has no formal parameters.

```
 1100   REM set sub$(buffer$,1,1) equal to $28 and set the rest of
 1110   REM the buffer to status table values
 1120   PRINT:PRINT"Not implemented"
 1125   errorcode=1
 1130   RETURN
```

This would normally be the "restore console status" function. Because it is both dangerous and undocumented, it is "left to the reader as an exercise." In general you should only restore with a buffer loaded during a previous "preserve console status" call (Status Code 1, above).

```
 1200   PRINT:INPUT"Do you want to terminate input with a specific
character?";a$
 1210   GOSUB 5000
 1215   IF NOT yes THEN 1250
 1220   INPUT"ASCII value of termination character: ";a$
 1225   char=0:char=CONV(a$)
 1230   IF char<1 OR char>255 THEN 1200
 1235   buffer$=CHR$(128)+CHR$(char)
 1240   RETURN
 1250   buffer$=CHR$(0)+CHR$(0)
 1255   RETURN
```

By contrast, control 2 above is nice and safe, and also serves to introduce the input test routine at line 5000, as follows:

```
 5000   yes=1
 5010   ans$=MID$(a$,1,1):IF ans$<>"Y" AND ans$<>"y" THEN yes=0
 5020   RETURN
```

One of the things you might enjoy with the termination character routine is to remember that the "Open Apple" key adds 128 to the value of a given key value.

Therefore, to change the function of the RETURN key to require "Open-Apple RETURN" is simply a matter of setting the termination character to ASCII 141 (128+13). Amaze your friends! Note also that Basic does not reset these values until you reboot, so if you set the value to something you can't type (or happen to forget), you're out of luck!

```
 1300   INPUT"Do you want to do two byte reads from the keyboard? ";a$
 1305   GOSUB 5000
 1310   IF yes THEN buffer$=CHR$(128):RETURN
 1320   buffer$=CHR$(0):RETURN
```

Two byte reads are a whole world by themselves. This is the way to set up your application to use the numeric keypad for special functions, to read the state of the "Apple" keys, etc., etc., etc. More information is in the Device Drivers manual. If interest is high enough in these techniques, maybe it would make the subject of a future column.

```
 1400   INPUT"Size of the type-ahead buffer: ";a$
 1405   IF a$="" THEN errorcode=1:RETURN
 1410   size=CONV(a$)
 1415   IF size<0 OR size>128 THEN PRINT"Invalid input, try again":GOTO
1400
 1420   buffer$=CHR$(size)
 1425   RETURN
```

Type-ahead buffer size is interesting only in the ability to set it to zero (no type-ahead).

```
 1500   buffer$=CHR$(0)
 1510   RETURN
```

This just flushes the type-ahead buffer (like CTRL-6) which is handy if you want to guarantee a certain input state or timing.

```
 1600   PRINT"Warning, Don't make a mistake!"
 1605   INPUT"Attention event priority: ";a$
 1610   IF a$="" THEN errorcode=1:RETURN
 1615   pri=CONV(a$):IF pri<0 OR pri>255 THEN 1605
 1620   buffer$=CHR$(pri)
 1625   INPUT"Attention Event ID: ";a$
 1630   IF a$="" THEN 1605
 1635   event=CONV(a$):IF event<0 OR event>255 THEN 1625
 1640   buffer$=buffer$+CHR$(event)
 1645   INPUT"Attention event handler address (three bytes): ";a$
 1650   IF LEN(a$)<>3 THEN 1645
 1655   buffer$=buffer$+a$
 1660   INPUT"ASCII code of Attention character: ";a$
 1665   code=CONV(a$):IF code<1 OR code>255 THEN 1660
```

```
 1670   buffer$=buffer$+CHR$(a$)
 1675   PRINT"Buffer is: ";
 1680   FOR i=1 TO 6:PRINTUSING"2a,x";MID$(HEX$(ASC(MID$
(buffer$,i,1))),3,2);:NEXT
 1685   INPUT"  Ok? ";a$:GOSUB 5000
 1690   IF yes THEN RETURN:ELSE:GOTO 1605
```

Here's where life gets dangerous. If you use this routine, you must first call attention status to get the priority, event id and address, and re-enter that information exactly. Only then can you feel free to change the attention character.

```
 1700   errorcode=1:RETURN
```

Control call number 7 is reserved.

```
 1800   PRINT"Warning, Don't make a mistake!"
 1805   INPUT"Any-key event priority: ";a$
 1810   IF a$="" THEN errorcode=1:RETURN
 1815   pri=CONV(a$):IF pri<0 OR pri>255 THEN 1805
 1820   buffer$=CHR$(pri)
 1825   INPUT"Any-key Event ID: ";a$
 1830   IF a$="" THEN 1805
 1835   event=CONV(a$):IF event<0 OR event>255 THEN 1825
 1840   buffer$=buffer$+CHR$(event)
 1845   INPUT"Any-key event handler address (three bytes): ";a$
 1850   IF LEN(a$)<>3 THEN 1845
 1855   buffer$=buffer$+a$
 1860   PRINT"Buffer is: ";
 1865   FOR i=1 TO 5:PRINTUSING"2a,x";MID$(HEX$(ASC(MID$
(buffer$,i,1))),3,2);:NEXT
 1870   INPUT"  Ok? ";a$:GOSUB 5000
 1875   IF yes THEN RETURN:ELSE:GOTO 1805
```

What was said for the attention event above goes double for the routine above.

```
 1900   errorcode=1:RETURN
```

Control call number 9 is reserved.

```
 2000   INPUT"Do you want No-Wait input? ";a$
 2005   IF a$="" THEN errorcode=1:RETURN
 2010   GOSUB 5000
 2015   IF yes THEN buffer$=CHR$(128):RETURN
 2020   buffer$=CHR$(0):RETURN
```

No wait input is fun to play with because it bypasses buffering and waiting for RETURN and gives the inputting program whatever has accumulated since the last input request. The program is responsible for making sense of the entered

characters. This might come in handy in conjunction with the console synchronizing function (chr$(22)) to wait a certain time and see what had been typed in that time period.

```
2100   INPUT"Do you want any echoing of characters to the screen? ";a$
2105   IF a$="" THEN errorcode=1:RETURN
2110   GOSUB 5000
2115   IF yes THEN 2130
2120   buffer$=CHR$(0)
2125   RETURN
2130   INPUT"Do you also want control characters to be echoed? ";a$
2135   IF a$="" THEN 2100
2140   GOSUB 5000
2145   IF yes THEN buffer$=CHR$(192):RETURN
2150   buffer$=CHR$(128)
2155   RETURN
```

This is a handy function for typing in passwords and other characters where selective display is desired. In addition, it allows programatic setting of the CONTROL-8 (display control characters) keyboard function.

```
2200   INPUT"Do you want the Retype function enabled? ";a$
2205   IF a$="" THEN errorcode=1:RETURN
2210   GOSUB 5000
2215   IF yes THEN buffer$=CHR$(128):RETURN
2220   buffer$=CHR$(0)
2225   RETURN
```

Retype allows the forward arrow key to copy characters into the buffer. This is the normal mode for Basic, but can be disallowed to provide pure cursor movement.

```
2300   INPUT"Do you want the Backspace function to be enabled? ";a$
2305   IF a$="" THEN errorcode=1:RETURN
2310   GOSUB 5000
2315   IF yes THEN 2330
2320   buffer$=CHR$(0)
2325   RETURN
2330   INPUT"Do you also want backspace to be destructive? ";a$
2335   IF a$="" THEN 2300
2340   GOSUB 5000
2345   IF yes THEN buffer$=CHR$(192):RETURN
2350   buffer$=CHR$(128)
2355   RETURN
```

This is really handy for applications programs taking input from the keyboard using standard input statements. It, in connection with disabling the retype

function, allows the naive user to be sure that what shows up on the screen is what is actually going to be input when he presses RETURN.

```
2400   INPUT"Do you want the Cancel function to be enabled? ";a$
2405   IF a$="" THEN errorcode=1:RETURN
2410   GOSUB 5000
2415   IF yes THEN 2430
2420   buffer$=CHR$(0)
2425   RETURN
2430   INPUT"Do you also want Cancel to be destructive? ";a$
2435   IF a$="" THEN 2400
2440   GOSUB 5000
2445   IF yes THEN buffer$=CHR$(192):RETURN
2450   buffer$=CHR$(128)
2455   RETURN
```

Turning on "destructive cancel" is the most powerful feature of this routine. By now you have probably gotten accustomed to getting a backslash and then a carriage return and line feed when you press CTRL-X. When destructive cancel is turned on, the console driver instead issues a destructive (erasing) backspace for every character in the input buffer. The net effect of this is that the cursor snaps back to the exact place it was when you began to type that line, erasing everything as it goes. This is not particularly handy if you are writing programs and want to cancel a line but then use the forward arrow to correct the mistake. However, it makes lots of sense in an application where the user might be confused by the backshash and the cursor sitting on the line below, expectantly waiting for who knows what.

```
2500   INPUT"Do you want Escape Mode enabled? ";a$
2505   IF a$="" THEN errorcode=1:RETURN
2510   GOSUB 5000
2515   IF yes THEN buffer$=CHR$(128):RETURN
2520   buffer$=CHR$(0):RETURN
```

Escape mode is another handy thing while you are developing programs which you might want to turn off during an application program, especially if you want to restrict the input to just the prompted locations. If you use destructive backspace and destructive cancel, it makes sense to disable escape mode also.

```
2600   errorcode=1:RETURN
```

Down-loading an entire character set is not implementable with this invokable because of the 255 character limit on the buffer. However, don't despair, since there is the great "download.inv" to take care of the problem. See your friendly "download.doc" file on the Basic product disk for more details.

```
2700   PRINT"Not currently implemented"
2705   REM sub$(buffer$,1,1) contains the character count
```

```
 2710   REM the rest is individual character definitions (max 8)
 2715   errorcode=1:RETURN
```

Even though it is not possible to download an entire character set using this invokable, control request 17 above does provide a mechanism for downloading eight characters at a time. Although the process for developing the character definitions is beyond the scope of this article, there is enough information in the Device Drivers manual to get you started.

```
 2800   errorcode=1:RETURN
```

The process of restoring a viewport is also left to the reader. Because of the 255 character buffer limit discussed earlier, it is not possible to restore a whole viewport. Further, the initial three byte code at the front of the buffer is undocumented, which makes it a little tough to create your own viewport definitions. Don't be discouraged, however. There's lots to do above which can make your application development smoother and more friendly (and lots of time required to test everything out!)

Homework!

This has been quite an treatise on the use of the Apple III console. There are lots of other useful options in the console driver that will be covered next time, but in the meanwhile, here is a present/homework assignment which should give you some chuckles while proving that there is very little you can't do with SOS and Basic. Without further ado, try out the following:

```
 5   INPUT"File to fill with assorted trash: ";file$
 10   IF file$="" THEN 230
 15   OPEN#1,file$
 20   INPUT"How many lines of this trash do you want to create: ";line$
 25   line=0:line=CONV(line$)
 30   IF line<1 THEN GOTO 5
 45   DATA "duke","prince","frog","sanitation engineer","dowager
duchess"
 50   DATA "captivated","impressed","repulsed","bored","completely
overwhelmed"
 55   DATA "handsome","pathetic","eager","reluctant","willing"
 60   DATA "wholesome","reserved","wild","enthusiastic","shy"
 65   DATA "king","queen","tadpole","flagpole climber","lady marine"
 70   DATA "mother","father","grand-parents","analyst","best friends"
 80   FOR i=1 TO 5:READ a$(i):NEXT
 90   FOR i=1 TO 5:READ b$(i):NEXT
 100   FOR i=1 TO 5:READ c$(i):NEXT
 110   FOR i=1 TO 5:READ d$(i):NEXT
 120   FOR i=1 TO 5:READ e$(i):NEXT
 130   FOR i=1 TO 5:READ f$(i):NEXT
```

```
 140   FOR i=1 TO line
 145     PRINT#1; USING"3#,x";i;
 150     PRINT#1;"Once a ";a$(INT(RND(1)*5+1));
 160     PRINT#1;" was ";b$(INT(RND(1)*5+1));
 170     PRINT#1;" by a ";c$(INT(RND(1)*5+1));
 180     PRINT#1;" but ";d$(INT(RND(1)*5+1));
 190     PRINT#1;" young ";e$(INT(RND(1)*5+1));
 200     PRINT#1;" which neither of them talked to their
";f$(INT(RND(1)*5+1));" about."
 210     NEXT i
 220   CLOSE
 230   END
```

Running the little jewel above should create the following kind of file (be sure to run at least 50 lines):

1 Once a dowager duchess was impressed by a reluctant but enthusiastic young flagpole climber which neither of them talked to their mother about.

2 Once a duke was completely overwhelmed by a pathetic but wholesome young tadpole which neither of them talked to their analyst about.

3 Once a duke was bored by a willing but wild young queen which neither of them talked to their grand-parents about.

4 Once a prince was repulsed by a pathetic but shy young tadpole which neither of them talked to their analyst about.

Unfortunately if you print this file out on the screen (by replying .console to the filename prompt, for example) you get the sentences wrapped around because the lines are more than 80 characters long. How nice it would be if you could see them printed out and scroll horizontally to read them, just as the console already scrolls down to allow more information to be printed. Next month we'll explore the following program in detail, and other programming tricks with the console, but for now, type and enjoy:

```
 1   DIM a$(500)
 5   INPUT"File name to scroll through: ";a$
 10   IF a$="" THEN 200
 15   OPEN#1,a$
 20   maxlenght=0
 25   ON EOF#1 GOTO 35
 30   FOR i=0 TO 500:INPUT#1;a$(i):IF LEN(a$(i))>maxlenght THEN
maxlenght=LEN(a
      $(i)):NEXT:ELSE:NEXT
 35   lastrecord=i
```

```
 40  leftscroll$=CHR$(23)+CHR$(255) +CHR$(26)+CHR$(79)+
CHR$(0)+CHR$(2)+CHR$(26)+CHR$(0)+CHR$(24)+CHR$(3)+CHR$(21)+"5"+
CHR$(12)+CHR$(22)
 45 rightscroll$=CHR$(23)+CHR$(1)+ CHR$(26)+CHR$(0)+CHR$(0)+
CHR$(2)+CHR$(26)+CHR$(0)+CHR$(24)+CHR$(3)+CHR$(21)+"5"+
CHR$(12)+CHR$(22)
 50  scrollup$=CHR$(16)+CHR$(3)+CHR$(26)+CHR$(0)+CHR$(23)+CHR$(10)+
CHR$(21)+"5"+CHR$(22)
 55   scrolldown$=CHR$(16)+CHR$(3)+CHR$(12)+CHR$(11)+CHR$(22)
 60   HOME:PRINT CHR$(21);"5";
 65   FOR i=0 TO 23:IF LEN(a$(i))>80 THEN PRINT MID$(a$
(i),1,80);:NEXT:ELSE PRINT a$(i):NEXT
 70   hi=1:vi=24:TEXT
 75   bnk$="                         ":b$=bnk$:c$=bnk$
 80   GET a$:cursor=ASC(a$)
 85   move=(cursor=8)+2*(cursor=21)+3*(cursor=10)+4*(cursor=11)
 90   ON move+1 GOSUB 130,100,105,110,120
 95   GOTO 80
 100   IF hi>1 THEN index=vi-25:hiindex=hi-1:FOR j=1 TO
24:SUB$(c$,j,1)=MID$(a$(index+j),hiindex,1):NEXT:PRINT
rightscroll$;c$;:hi=hi-1:c$=bnk$:TEXT
 102   RETURN
 105   IF hi+80<=maxlenght THEN index=vi-25:hiindex=80+hi:FOR j=1 TO
24:SUB$(b$,j,1)=MID$(a$(index+j),hiindex,1):NEXT:PRINT leftscroll$;b$;:
hi=hi+1:b$=bnk$:TEXT
 107   RETURN
 110   IF vi<lastrecord THEN PRINT scrollup$;MID$(a(vi),hi,80);:
vi=vi+1:TEXT
 115   RETURN
 120   IF vi>24 THEN PRINT scrolldown$;MID$(a$(vi-25),hi,80);:
vi=vi-1:TEXT
 125   RETURN
 130   IF cursor=27 THEN POP:GOTO 200:ELSE:RETURN
 200   TEXT:PRINT CHR$(26);CHR$(0);CHR$(23);
 210   CLOSE
 220   END
```

Be sure you type it in exactly as written, and be sure that "bnk$" in line 75 contains exactly 24 spaces between the quote marks. Then use the first program to create a file of junky messages. If you want, you can change the data statements to make up your own messages. Then use that same file name for the second program and use the arrow keys to scroll up, down, left and right through the file. If you have trouble, just wait until next month when we clear some of the murky water.

Until then, there doesn't seem to be anything else that can console you, so have fun with your Apple III!

Exploring Business Basic, Part XI

Last article we dug down deep in the "SOS mines" to find a number of useful things in the .CONSOLE driver that can make developing interesting applications a lot easier and more efficient. At the end of that article, which you should really read before this one if possible, a parting challenge was given in the form of a program listing without documentation or explanation. This time we will explore that program, and go even further into uses of the console features in constructing business oriented applications programs.

Digging Out

The mysterious program at the end of last month's article was designed to allow you to do four-way scrolling through text files by using a number of console features, especially windowing and horizontal scrolling. Rather than describe that previous program, however, here's a new version with even better features (and some simplification).

```
1   DIM a$(500),rightscroll$(1),leftscroll$(1)
5   INPUT"File name to scroll through: ";a$
10   IF a$="" THEN 200
15   OPEN#1,a$
20   maxlenght=0
22   INPUT"How many units to fast scroll by? ";zip
```

Last article we dug down deep in the "SOS mines" to find a number of useful things in the .CONSOLE driver that can make developing interesting applications a lot easier and more efficient. At the end of that article, which you should really read before this one if possible, a parting challenge was given in the form of a program listing without documentation or explanation. This time we will explore that program, and go even further into uses of the console features in constructing business oriented applications programs.

Digging Out

The mysterious program at the end of last month's article was designed to allow you to do four-way scrolling through text files by using a number of console features, especially windowing and horizontal scrolling. Rather than describe that previous program, however, here's a new version with even better features (and some simplification). The program starts with some initialization lines. The string array "a$" is used to hold the contents of the text file. Using the disk directly is possible, but presents some difficulties which would obscure the real intent of the program. "Rightscroll$" and "leftscroll$" are string arrays which contain two versions of the scrolling commands for horizontal scrolling. One

version will do a column at a time, and the other multiple columns, determined by the variable "zip".

```
 25   ON EOF#1 GOTO 35
 30   FOR i=0 TO 500:INPUT#1;a$(i):IF LEN(a$(i))>maxlenght THEN
maxlenght=LEN(a$(i)):NEXT:ELSE:NEXT
 35   lastrecord=i
```

The lines above read in the contents of the file into "a$" and set the values of "maxlenght" (used to set the rightmost limit) and "lastrecord" (used to set the bottom limit).

```
 37   sync$=CHR$(22)
 40 leftscroll$(0)=SYNC$+CHR$(23)+CHR$(256-zip)+ CHR$(26)+CHR$(80-zip)
+CHR$(0)+CHR$(2)+CHR$(26)+CHR$
(zip-1)+CHR$(24)+CHR$(3)+CHR$(21)+"5"+CHR$(12)
 41 leftscroll$(1)=SYNC +CHR$(23)+CHR$(255)+CHR$(26)+CHR$(79)+
CHR$(0)+CHR$(2)+CHR$(26)+CHR$(0)+CHR$(24)+CHR$(3)+CHR$(21)+"5"+CHR$(12)
 45  rightscroll$(0)=SYNC$+CHR$(23)+CHR$(zip+CHR$(26)+CHR$(0)+
CHR$(0)+CHR$(2)+CHR$(26)+CHR(zip-1)+CHR$(24)+CHR$(3)+CHR$(21)+
"5"+CHR$(12)
 46 rightscroll$(1)=SYNC+CHR$(23)+CHR$(1)+CHR$(26)+CHR$(0)+CHR$(0)+
CHR$(2)+CHR$(26)+CHR$(0)+CHR$(24)+CHR$(3)+CHR$(21)+"5"+CHR$(12)
 50 scrollup$=CHR$(16)+CHR$(3)+CHR$(26)+CHR$(0)+CHR$(23)+CHR$(10)+
CHR$(21)+"5"+SYNC$
 55   scrolldown$=CHR$(16)+CHR$(3)+CHR$(12)+CHR$(11)+SYNC$
```

The lines above set up the scrolling commands to be issued to the console driver. It would be useful to follow along in your Standard Device Drivers manual in the section on the .CONSOLE driver, but here goes one example to help you through it:

Line 41 defines the character codes necessary to set up scrolling horizontally to the left, one character position at a time. Here's how the character sequence works. First comes the sync character (decimal 22) which is used to start the action in step with the screen refresh. This helps to eliminate flicker. Next, character 23 is the command to horizontally shift the screen. The following character (255) determines that the shift is one character position to the left. This action leaves the rightmost column on the screen blank, which will later get filled with information previously off the screen. To create a place to put this data, the next two sets of commands create a window one column wide and 24 lines high in which to do the writing. Character sequence 26,79,0 addresses the cursor to column 79, line 0, and character 2 establishes this position as the upper right corner of a "window". Since cursor addressing is done relative to the window, the next command sequence (characters 26,0,23) puts the cursor at the bottom of this one-column window. There, printing a character 3 establishes the position as the lower right corner of the window. From now on, this will be the only console area which can be written to. The next two characters (decimal 21

and the literal "5") set up cursor movement options which will allow us to write data into this window very rapidly, since the "5" option has the effect of turning off scrolling and new-line, while leaving "wrap" and cursor advance intact. This means that if a single string is written into this window, the characters will spill down the screen, one per line. One 24-character string could fill the entire window, written with one print statement. This is much, much faster than individually positioning the cursor and then printing the characters one at a time. The last character in the string in line 41 (the form-feed character, decimal 12) serves to home the cursor to the top left of the window (in this case, column 79, line 0) preparing us to print the missing string which will fill the blank space created by the horizontal shift. Notice also that line 40 is nearly identical, except that the position of the top left and bottom right of the window are determined by the value of "zip".

Scrolling to the right is handled in the same way, by setting up characters in "rightscroll". Check through those characters to make sure you understand what is happening.

```
 60   HOME:PRINT CHR$(21);"5";
 65   FOR i=0 TO 23:IF LEN(a$(i))>80 THEN PRINT MID$(a$
(i),1,80);:NEXT:ELSE PRINT a$(i):NEXT
```

The two lines above disable vertical scrolling and put the first 24 lines of the text file on the screen. Note that care is taken to put only the first 80 characters of each line on the screen, and that by disabling scroll, it is possible to write the screen completely full.

```
 70   hi=1:vi=24:TEXT
 72   blank24$="                          "
 73   bnk$=""
 75   FOR i=1 TO zip:bnk$=bnk$+blank24$:NEXT
```

These lines do further initialization. The variables "hi" and "vi" stand for horizontal and vertical indexes, which tell the program where in the text file is the lower lefthand corner of the screen. "Blank24$" and "bnk$" are areas to store data to be written rapidly to the screen, for single column scroll and multiple column scroll respectively.

```
 80   GET a$:cursor=ASC(a$)

 85  move=(cursor=8)+2*(cursor=21)+3*(cursor=10)+4*(cursor=11)+5*
(cursor=136)+6*(cursor=149)

 88   b$=blank24$:t=1
 90   ON move+1 GOSUB 130,100,105,110,120,135,140
 95   TEXT:GOTO 80
```

The section above represents the major loop of the program. Here the cursor commands are accepted in line 80, and decoded into a command number in line 85. Be sure you understand how line 85 creates values using the logical

expressions. This is really handy (Pascal users have something similar in the CASE statement). Once the "move" variable is calculated, then an ON ... GOSUB statement is used to go to the appropriate subroutine. Note that values 1,2,3,4 of "move" correspond with left, right, down and up on the cursor keys. Values 5 and 6 represent the left and right arrow keys with the "open-Apple" key pressed. Remember that "open-Apple" adds 128 to the ASCII value of the key. This trick is going to be used to implement the "zip" mode we defined above, just watch!

```
 100   IF hi>t THEN index=vi-25:hiindex=hi-t:FOR j=1 TO
24:SUB$(b$,t*j-t+1,t)=MID$(a$(index+j),hiindex,t):NEXT:hi=hi-t:PRINT
rightscroll$((t=1));b$;
 102   RETURN
 105   IF hi+80<=maxlenght THEN index=vi-25:hiindex=80+hi:FOR j=1 TO
24:SUB$(b$,t*j-t+1,t)=MID$(a$(index+j),hiindex,t):NEXT:hi=hi+t:PRINT
leftscroll$((t=1));b$;
 107   RETURN
```

Here's where things get a bit sticky. Remember we said that the technique for scrolling was to use the console horizontal shift and then fill in the empty space with the appropriate characters from the file. Here then are the routines which do the horizontal. Line 100 implements the left-arrow function by first checking to see if the horizontal index ("hi") is greater than the left shift ("t") required. If everything is ok, then "index" is set to the top of the 24 lines of text to be scrolled, and the new left edge is calculated and assigned to "hiindex". Then a loop in this same line fills "b$" with the appropriate contents of "a$", the array containing the file contents. The SUB$ function is used to increase the performance of this loop, by directly substituting characters in an existing string. With this complete, the horizontal index is adjusted by the width of the scroll, and then a print statement directs the special characters required to do the horizontal shift. Note that another logical expression (t=1) is used to pick the appropriate string from the two defined above. Immediately after the scrollstring is printed, the "b$" string is printed, which spills down the vertical window the characters previously extracted from the text array. By using this technique, it is possible to scroll so fast as to not appear to actually be writing characters on the screen. The only slow part of this routine, in fact, is in the loop which creates the string "b$" to be written.

The scroll routine in line 105 is similar, except for the fact that it must first check to be sure that scrolling doesn't occur past the end of the longest line (previously calculated as "maxlenght").

The routines above may not become clear with out pencil and paper and some diagrams, and appologies are offered for the obscure way that all the statements are crammed onto one line. However, the consequence of breaking everything out neatly was considerably worse performance, and besides, after all these articles, you can probably make sense out of anything.

```
110   IF vi<lastrecord THEN PRINT scrollup$;MID$(a$(vi),hi,80);:vi=vi+1
115   RETURN
120   IF vi>24 THEN PRINT scrolldown$;MID$(a$(vi-25),hi,80);:vi=vi-1
125   RETURN
```

Lines 110 and 120 are vertical scrolling, and therefore considerably easier.

After checking to be sure that scrolling is allowed, the scroll characters are printed and the appropriate substring is printed to the screen. Note that the MID$ function makes selecting 80 characters from the string very easy.

```
130   IF cursor=27 THEN POP:GOTO 200
132   RETURN
```

Line 130 handles the case of stray characters, and uses the "Escape" key (ASCII 27) as the way out of the routine.

```
135   IF hi>zip THEN t=zip:b$=bnk$
137   GOSUB 100:RETURN
140   IF hi+79+zip<=maxlenght THEN t=zip:b$=bnk$
142   GOSUB 105:RETURN
```

Line 135 and 140 handle the case of the "Open-Apple" cursor keys. Note that if the full "zip" increment on cursor movement is not possible, the routine reverts to the initial conditions, a horizontal shift of one, set in line 88.

After setting the appropriate value, a gosub is executed to the main scroll subroutine.

```
200   TEXT:PRINT CHR$(26);CHR$(0);CHR$(23);
210   CLOSE
220   END
```

Last but not least, wrapup and conclusion, positioning the cursor to the bottom of the screen and terminating the program.

Well, here's hoping that you have lots of fun playing with this program and the various text files you have laying around. One note of caution, however. This program because of the Basic limit of 255 characters maximum in a string, will not work with all text files. Among the examples are many Applewriter III files, since it is easy to create enormous amounts of text without benefit of intervening carriage return characters. If you want to test the program with Applewriter or similar text files, you'll want to first print them out to disk (instead of simply saving them) and then load them into the scroll program. This works equally well with Visicalc print files, as long as the width is not over 255 characters.

In addition, it you really want a text file to fool with, check back to last month's article, where a gibberish-generating program was described which is guaranteed to produce interesting things to scroll through. Some people have claimed that it can be used to create this column as well, but lining up that

infinite number of monkeys at an infinite number of Apple IIIs has some details still to be worked out. Oh, well...

Something Useful from All This

Although scrolling on the screen is useful, and even fun, there are other, far more typical uses of the console features that most applications could use. One of the most common of these is the use of data entry screens. Everybody who has programmed has wished for easy ways to generate the data entry screens which are an inevitable part of any business application. Most programmers sooner or later create or buy some software to make that task easier. Not to be outdone, your fearless Basic columnist offers the following tender morsel:

Nope, not so fast. First the sales pitch... The program below is generally organized along the following lines: first, a skeleton program which performs a general data entry loop of presenting a screen with fields to be filled in and second, a series of support subroutines which you could use to initialize a screen definition, present the screen, capture the data, and store it in a transaction file. In addition to these functions, the routines are designed in such a way as to allow quite a bit of flexibility in adding features of you own design, especially edit routines on the data.

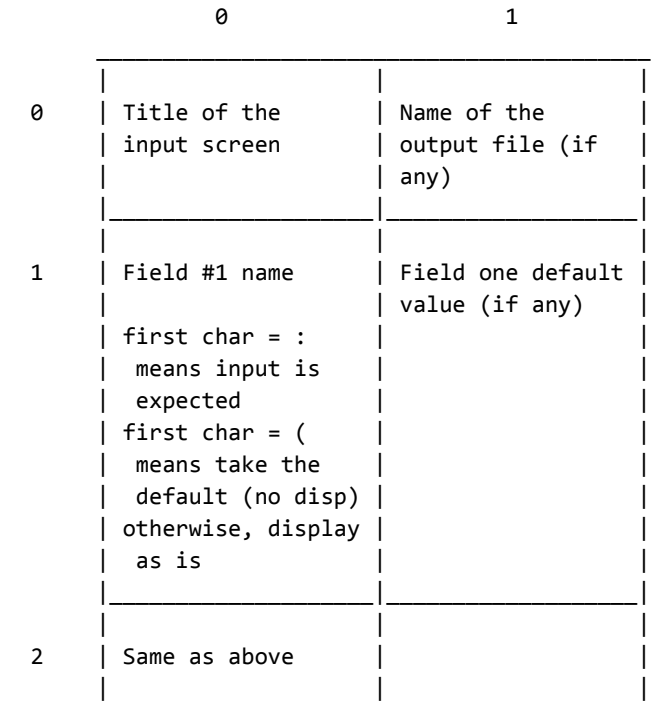
Ok, now that the orientation is over, here's the program skeleton:

```
 1   REM Screen data capture program
 5   DIM name$(50,1),info%(50,2),input.req%(50)
 20   GOSUB 1000
 25   HOME
 30   PRINT:PRINT"Data Entry for Screen: ";screen$
 35   IF writefile THEN PRINT:PRINT"with output stored in the file
";outfile$
 40   VPOS=23:HPOS=1:PRINT"Press any key to begin:";
 45   GET a$
 100   FOR recordnum=1 TO 32767
 105     GOSUB 1500:REM display the data entry screen with defaults
 107     escapecode=0:out.rec$=""
 110     FOR fieldnum=1 TO items
 115       GOSUB 2000:REM process input for field=fieldnum
 120       IF escapecode THEN IF fieldnum=first.input THEN TEXT:GOTO
600:ELSE:GOTO 105
 125       REM extra processing for this field goes here
 200       GOSUB 3000:REM add to output record string
 205       NEXT fieldnum
 210     REM code to process the finished record in outrec$ goes here
 215     TEXT:HOME:PRINT"Record is: ";out.rec$
 220     PRINT"Press any key to continue: ";:GET a$
```

```
 500     IF writefile THEN GOSUB 4000
 505     NEXT recordnum
 600   TEXT:HOME
 605   PRINT:PRINT"End of Data Entry for Screen: ";screen$
 610   IF writefile THEN PRINT:PRINT"Output is stored in the file
";outfile$
 615   VPOS=23:HPOS=1:PRINT"Press any key to quit:";
 620   GET a$
 630   CLOSE
 635   END
```

That's a fairly meaty skeleton, but relatively straightforward. First should come a word about the three arrays dimensioned in line 5. Since this is a general purpose data entry routine, all the information about the data to be captured is contained in arrays in memory.

"Name$" holds the name of each field to be displayed, along with any default values, in the following format:



etc.

"Info%" is an array which contains information about how the field names and values are to be displayed, to wit:

	0	1	2
0	lenght of output record	not used	not used
1	Starting row for field #1	starting column for field #1	lenght flag for field #1 if + then value is the maximum permitted if 0 then the field has no maximum value if - then value is required field lenght
2	more of the same for field number 2	likewise	onward, ever upward

etc.

The last array, "input.req%", is considerably simpler. It is built during initalization, and contains 1 if the field requires input, 0 if no input (titles, etc.) and a -1 if the field consists of a default value only.

Next the program performs a GOSUB to an initialization routine at line 1000. This routine, in addition to filling the three arrays mentioned above, also sets a number of constants and opens the data logging file, if indicated. The routine below uses initialization from DATA statements, but, as indicated, a "real" program would use files to contain the screen definitions.

```
1000   REM initialize tables (could be done from a file)
1005   first.input=0
1007   READ items
1010   FOR i=0 TO items
1015     READ name$(i,0),name$(i,1)
```

```
 1017     IF MID$(name$(i,0),1,1)=":" THEN
input.req%(i)=1:first.input=i*(first.input=0)+first.input
 1018     IF MID$(name$(i,0),1,1)="(" THEN input.req%(i)=-1
 1020     FOR j=0 TO 2:READ info%(i,j):NEXT j
 1025     NEXT i
 1030   screen$=name$(0,0):outfile$=name$(0,1)
 1035   outlen=info%(0,0)
 1040   IF outfile$="" THEN writefile=0:GOTO 1055:ELSE:writefile=1
 1045   OPEN#2,outfile$,outlen
 1055   set.edit$=CHR$(21)+"0"
 1060   set.normal$=CHR$(21)+"1"
 1062   REM blank$ below contains 80 space characters
 1065   blank$="                                                                "
 1095   RETURN
```

Of passing interest in the routine above is the use in line 1017 of a logical expression to put the index of the first field requiring input in the variable "first.input". This could have been done nearly as easily with an IF statement, but there's a special on logic this week that seemed too good to pass up. "First.input" itself is used to determine when pressing "escape" should mean stop inputting, as opposed to just starting the current screen over.

Of more than passing interest is looking as a sample set of screen definitions which this program might process. Consider the following DATA statements as an example:

```
 1700   DATA 7
 1705   DATA "My First Screen",""
 1707   DATA 117,0,0
 1710   DATA "Name and Address Entry",""
 1715   DATA 1,30,0
 1720   DATA ":First Name: ",""
 1730   DATA 3,1,15
 1735   DATA ":Last Name: ",""
 1740   DATA 3,40,20
 1745   DATA "Address (free form)",""
 1750   DATA 5,1,0
 1755   DATA ":",""
 1760   DATA 6,1,0
 1765   DATA ":State: ","CA"
 1770   DATA 8,1,-2
 1775   DATA "(","FY1982"
 1780   DATA 0,0,0
```

The definition starts with the number of screen items (both displayable and not) and the next two lines are the general screen definition. Next comes a sample

screen comment ("Name and Address Entry") which line 1715 tells us will be positioned on row 1, beginning at column 30. The next field requires input (the leading ":" indicates that) and has no default value, and lives on row 3 column 1. Furthermore, it has a maximum allowed lenght of 15 characters. Line 1745 is another comment, this one refering to the field directly under it and defined on lines 1755 and 1760. Since this is a free-form field, with no title (the ":" is its only definition) it will extend the entire lenght of the line, a full 80 characters of input space. Line 1765 is an example of a field with a default value, and also one (as indicated in line 1770) which has a required lenght of two characters. The last example, on line 1775, is a default field which will appear in all output records. This is a useful option for including fields like dates, header data, etc. which the user should not be required to type each time, but which may need to appear in the output for reference or to meet another program's requirements.

That about wraps up the initalization, leaving us with a set of screen and input definitions for a simple data entry screen. Now let's go back and look at the rest of the program main loop, starting with line 25. Here and through line 45 we create a starter screen, which certainly could be more elaborate if wished. One thing that comes to mind is to prompt here for the name of the screen definition file instead of hard-coding it as was done in this example.

In any case, line 100 begins the program's main loop for data entry. The first routine called is the subroutine at line 1500 which displays the screen according to the definitions. It looks like this:

```
1500   TEXT:HOME
1505   FOR field=1 TO items
1510     field$=name$(field,0)
1515     IF MID$(field$,1,1)=":" THEN 1550
1520     IF MID$(field$,1,1)="(" THEN 1600
1525     VPOS=info%(field,0):HPOS=info%(field,1)
1530     PRINT name$(field,0);
1535     GOTO 1600
1550     VPOS=info%(field,0):HPOS=info%(field,1)
1555     PRINT MID$(field$,2,LEN(field$)-1);
1560     IF name$(field,1)="" THEN 1600
1565     PRINT name$(field,1);
1600     NEXT field
1605   RETURN
```

If you have followed along in the discussion about field definition, the routine above should prove very straight-forward.

The next major task of our main program loop occurs at line 110, where an inner loop starts which process input from each field on the screen, one field at a time. This is accomplished in the subroutine at line 2000, and here's where things get a trifle tricky:

```
 2000   field=fieldnum
 2002   value$=""
 2005   IF input.req%(field)=0 THEN RETURN
 2006   IF input.req%(field)<0 THEN value$=name$(field,1):RETURN
```

These first few lines are fairly obvious. Using the "input.req%" array, we can quickly determine if the field is one requiring no or only default processing. Note that the string "value$" will be used to convey the result of this field's data entry process. Once we determine that actual input must take place, then the real work begins, as shown below:

```
 2008   row=info%(field,0)
 2010   start.window=info%(field,1)+LEN(name$(field,0))-1
 2015   field.len=ABS(info%(field,2))
 2020   IF field.len<>0 THEN end.window=start.window+field.len-1:GOTO
2060
 2025   test=field+1
 2030   IF test>items OR info%(test,0)>row THEN
end.window=79:field.len=80-start.window:GOTO 2060
 2035   IF info%(test,0)=row THEN
end.window=info%(test,1)-1:field.len=end.window-start.window+1:GOTO 2060
 2040   test=test+1
 2045   GOTO 2030
 2060   WINDOW start.window,row TO end.window,row
```

This routine sets up a field for data entry. Because of the console driver's powerful windowing capability, it is possible, once the size of the field is determined, to construct a cell on the screen for each data item which must be input. As you can see, this window definition is relatively easy if the field lenght is known up front. Lines 2025 through 2045 are designed to determine the actual field length available to a variable length item, by looking ahead at what's next on the screen and adjusting accordingly. Once that is determined, line 2060 establishes the window in which data entry will take place for that item. Next is the printing of any default values, and the display (in inverse) of the field available for entry:

```
 2065   line$=MID$(blank$,1,field.len)
 2070   default$=name$(field,1):IF default$<>"" THEN
SUB$(line$,1,LEN(default$))=default$
 2075   INVERSE:HOME
 2080   PRINT line$;
 2082   PRINT set.edit$;
 2085   HPOS=1:point=1
```

Of note here is the variable "set.edit", which is used to turn off all console options, leaving the program totally in control of cursor movement, wrap, scroll, etc. Line 2085 then positions the cursor to the beginning of the field (remember

that this is a window now) and sets up a pointer "point" to the first character of the field value ("line$"). Now the fun really begins:

```
 2100   ON KBD GOTO 2200
 2105   NORMAL:PRINT MID$(line$,point,1);:INVERSE:FOR j=1 TO
150:NEXT:PRINTMID$(line$,point,1);:FOR j=1 TO 150:NEXT:GOTO 2105
```

This is our old friend the ON KBD loop. In this case we are using the NORMAL and INVERSE options, and the fact that we just turned the console "advance after printing" function off, to blink whatever character in "line" string we are currently pointing at. For the purposes of this routine, you can equate "line$" and what's seen in the window exactly. Of course, you hum around in the little loop in line 2105 until a key is pressed. That sends the program off to line 2200:

```
 2200   OFF KBD
 2205   IF KBD<32 OR KBD>127 THEN 2270
 2210   SUB$(line$,point,1)=CHR$(KBD)
 2215   INVERSE:PRINT MID$(line$,point,1);
 2220   IF point<field.len THEN point=point+1
 2250   HPOS=point
 2255   ON KBD GOTO 2200
 2260   RETURN
```

After checking for control or special function characters in line 2205, the typed character is inserted into the "line$" string at the current cursor position, and the character is reprinted in inverse to be sure that the ON KBD routine wasn't exited in the wrong state. Assuming there is room in the window, lines 2220 and 2250 update the pointer and advance the cursor to the new position. Then lines 2255 and 2260 clean up and return to the "blink" routine, awaiting another keystroke. But what about those special characters? Wait no longer:

```
 2270   IF KBD=27 THEN escapecode=1:POP:RETURN
 2275   IF KBD=8 AND point>1 THEN INVERSE:PRINT
MID$(line$,point,1);:point=point-1:GOTO 2330
 2280   IF KBD=21 AND point<field.len THEN INVERSE:PRINT
MID$(line$,point,1);:point=point+1:GOTO 2330
```

These lines check for "escape" and exit back to the calling level (the POP gets us out of the ON KBD routine and back to reality). In addition, lines 2275 and 2280 process the cursor keys for left and right arrow, first reprinting the current character and then resetting the pointer.

```
 2300   IF KBD=13 THEN
value$=MID$(line$,1,point-1):line$=MID$(value$,1,field.len):GOTO 2320
 2305   IF KBD<>141 AND KBD<>9 THEN 2350
 2310   value$=line$
 2320   NORMAL:HOME:PRINT set.normal$;line$;
 2325   POP:RETURN
```

```
2330   HPOS=point
2350   ON KBD GOTO 2200
2355   RETURN
```

These lines wrap up the routine once the user is satisfied that the field is complete. There are several options to signal completion. First, line 2300 processes the "Return" key, discarding anything to the right of where the return key was pressed. "Value$" is set to what's left, and "line$" is re-defined so that the actual data can be displayed in line 2320. Line 2305 processes the other option, full entry of whatever is in the window, no matter where the cursor is. As you can see, this occurs when either "Open-Apple Return" or "Tab" is pressed. "Set.normal$" turns advance back on, so that the value can be printed back into the window, this time with inverse off, to indicate that data entry is finished in that field.

All of that excitement leads us back to the main loop, now at line 120, where the result of the field call is analysed. If "escape" was pressed, a further check is made to see if it was pressed during the first input field of the form. If so, that is the indication to terminate input of forms, and the program jumps out of the loop. If "escape" is pressed in any other field, processing starts over at line 105 with a clean slate. Assuming the return was normal, with data for the field in "value$" then there is an opportunity to do any additional processing required and then add "value$" to the accumulating "out.rec$" in the routine at line 3000. After all the fields are processed, lines 215 and 220 display it and if the file logging option was set originally, the subroutine at line 4000 writes the result in a file. Here are simple examples of what these routines could look like:

```
3000   IF LEN(value$) THEN out.rec$=out.rec$+value$
3005   RETURN

4000   PRINT#2,recordnum;out.rec$
4010   RETURN
```

Obviously, "real" data entry programs will have much more elaborate processing and editing functions built in. This example was only a guide to how you might incorporate these techniques into your own programs.

Some things you might want to try in order to improve the program could include expanding the "info%" array to contain more information about editing (like: is the data alphabetic or numeric? Does it have a fixed decimal place? Can it have a null value, or must some non-blank or non-zero value be used?) For fixed record layout output (like simulating records on a keypunch machine, yuck!) you might want to add fields to define where in the output record the value is to be placed (starting byte and lenght, for example). If you are really clever, you can modify the routine to accept multi-line fields. All these and more ideas will probably occur to you as you work with the routine. Remember also, there is nothing sacred about any of the beginning of the program either. The

subroutines could just as easily be used within a completely different environment to support a program's screen-handling needs.

One last challenge

Last article we covered a lot of esoteric goodies in the console driver. One of the least understood, but most powerful capabilities is the "two byte read", where you can programmatically get a second byte which among other things can indicate whether or not "Enter" or "Return" has been pressed. Normally these two different keys both return the same ASCII code, but the data entry routines above cry out for that distinction, which we provided in this article by using "Tab" and "Open-apple Return". It's not easy to figure this one out, and some substantial re-writing of the field routines may be required, but I'll offer a copy of Quickfile III to the earliest postmarked solution. Send a listing to Softalk marked "The Third Basic Solution", and the winning routine will be published as soon as possible in these pages.

Keep koding with the kool konsole...

Exploring Business Basic, Part XII

In recent months, we've been digging up uses for the various features of the console driver. While last month's four-way scrolling program was valuable, and even fun, there are other, far more typical ways that most applications can use the console features One of the most common of these is the use of data entry screens. Anyone who has programmed has wished for easy ways to generate the data entry screens that are an inevitable part of any busines application. Most programmers sooner or later create or buy software to make that task easier. Not to be outdone, your fearless Basic columnist offers the following tender morsel.

(Nope, not so fast; first the sales pitch. The program below is generally organized along the following lines: first, a skeleton program that performs a general data entry loop, presenting a screen with fields to be filled in; and second, a series of support subroutines that you can use to initialize a screen definition, present the screen, capture the data and store it in a transaction file. In addition to these functions, the routines are designed to allow quite a bit of flexibility in adding features of your own design, especially edit routines on the data.) Okay, now that the orientation is over, here's the program skeleton:

```
 1   REM Screen Data Capture Program
 5   DIM name$(50,1),info%(50,2),input.req%(50)
 20   GOSUB 1000
 25   HOME
 30   PRINT:PRINT"Data Entry for Screen: ";screen$
 35   IF writefile THEN PRINT:PRINT"with output stored in the file
";outfile$
 40   VPOS=23:HPOS=1:PRINT"Press any key to begin:";
 45   GET a$
 100   FOR recordnum=1 TO 32767
 105     GOSUB 1500:REM Display the data entry screen with defaults
 107     escapecode=0:out.rec$=""
 110     FOR fieldnum=1 TO items
 115       GOSUB 2000:REM process input for field=fieldnum
 120       IF escapecode THEN IF fieldnum=first.input THEN TEXT:GOTO
           600:ELSE:GOTO 105
 125       REM Extra processing for this field goes here
 200       GOSUB 3000:REM add to output record string
 205       NEXT fieldnum
 210     REM code to process the finished record in outrec$ goes here
 215     TEXT:HOME:PRINT"Record is:      ";out.rec$
 220     PRINT"Press any key to continue:";:GET a$
 500     IF writefile THEN GOSUB 4000
```

```
 505     NEXT recordnum
 600   TEXT:HOME
 605   PRINT:PRINT"End of Data Entry for Screen: ";screen$
 610   IF writefile THEN PRINT:PRINT"Output is stored in the file
";outfile$
 615   VPOS=23:HPOS=1:PRINT"Press any key to quit:";
 620   GET a$
 630   CLOSE
 635   END
```

That's a fairly meaty skeleton, but relatively straight forward. First, a word about the three arrays dimensioned in line 5. Since this is a general-purpose data entry routine, all the information is contained in arrays in memory.

Name$ holds the name of each field to be displayed, along with any default values, in the format shown in figure 1. Info% is an array that contains information about how the field names and values are to be displayed, as shown in figure 2. The last array, input.req%, is considerably simpler. It is built during initialization and contains 1 if the field requires input, 0 if no input (titles and so on), and a -1 if the field consists of a default value only.

Next the program performs a gosub to an initialization routine at line 1000. This routine, in addition to filling the three arrays just mentioned, also sets a number of constants and opens the data logging file, if indicated. The routine below uses initialization from data statements, but, as indicated, a "real" program would use files to contain the screen definitions.

```
 1000   REM initialize tables (could be done from a file)
 1005   first.input=0
 1007   READ items
 1010   FOR i=1 TO items
 1015     READ name$(i,0),name$(i,1)
 1017     IF MID$(name$(i,0),1,1)=":" THEN input.req%(i)=i:first.input=i
 1018     IF MID$(name$(i,0),1,1)="(" THEN input.req%(i)=-1
 1020     FOR j=0 TO 2:READ info%(i,j):NEXT j
 1025     NEXT i
 1030   screen$=name$(0,0):outfile$=name$(0,1)
 1035   outlen=info%(0,0)
 1040   IF outfile$="" THEN writefile=0:GOTO 1055:ELSE:writefile=1
 1045   OPEN#2,outfile$,outlen
 1055   set.edit$=CHR$(21)+"0"
 1060   set.normal$=CHR$(21)+"1"
 1062   REM blank$ below contains 80 space characters
 1065   blank$="                                                                "
 1095   RETURN
```

BARGAIN BASEMENT LOGIC

Of passing interest in this routine is the use in line 1017 of a logical expression to put the index of the first field requiring input in the variable first.input. This could have been done nearly as easily with an IF statement, but there's a special on logic this week that seemed too good to pass up. First.input itself is used to determine whether pressing escape should mean stop inputting or just start the current screen over.

Of more than passing interest is a sample set of screen definitions that this program might process. Consider the following data statements as an example:

```
1700 DATA 7
1705 DATA "My First Screen",""
1707 DATA 117,0,0
1710 DATA "Name and Address Entry",""
1715 DATA 1,30,0
1720 DATA ":First Name: ",""
1730 DATA 3,1,15
1735 DATA ":Last Name: ",""
1740 DATA 3,40,20
1745 DATA "Address (free form)",""
1750 DATA 5,1,0
1755 DATA ":",""
1760 DATA 6,1,0
1765 DATA ":State: ","CA"
1770 DATA 8,1,-2
1775 DATA "(","FY1988"
1780 DATA 0,0,0
```

The definition starts with the number of screen items (both displayable and not) and the next two lines are the general screen definition. Next comes a sample screen comment ("name and address entry") which line 1715 tells us will be positioned on row 1, beginning at column 30. The next field requires input (the leading colon indicates that), has no default value, and lives on row 3, column 1. Furthermore, it has a maximum allowed length of fifteen characters. Line 1745 is another comment, this this one referring to the field directly under it snd defined on lines 1755 and 1760. Since this is a free-form field with no title (the colon is its only definition), it will extend the entire length of the line, a full eighty characters of input space.

Line 1765 is an example of a field with a default value, and also one (as indicated in line 1770) that has a required lenght of two characters. The last example, on line 1775, is a default field that will appear in all output records. This is a useful option for including fields, such as dates or heading data, that

the user should not be required to type each time, but that may need to appear in the output for reference or for meeting another program's requirements.

That about wraps up the initialization, leaving us with a set of screen and input definitions for a simple data entry screen. Now lets go back and look at the rest of the program main loop, starting with line 25. Here and through line 45 we create a starter screen, which could certainly be more elaborate if desired. For instance, you could prompt here for the name of the screen definition file instead of hard codeing it as we did in this example. In any case, line 100 begins the program's main loop for data entry. The first routine called is the subroutine at line 1500, which displays the screen according to the definitions. it looks like this:

```
1500   TEXT:HOME
1505   FOR field=1 TO items
1510     field$=name$(field,0)
1515     IF MID$(field$,1,1)=":" THEN 1550
1520     IF MID$(field$,1,1)="(" THEN 1600
1525     VPOS=info%(field,0):HPOS=info%(field,1)
1530     PRINT name$(field,0);
1535     GOTO 1600
1550     VPOS=info%(field,0):HPOS=info%(field,1)
1555     PRINT MID$(field$,2,LEN(field$)-1);
1560     IF name$(field,1)="" THEN 1600
1565     PRINT name$(field,1);
1600     NEXT field
1605   RETURN
```

If you have followed the discussion about field definition, the routine above should prove very straightforward.

The next major task of our main program loop occurs at line 110, where an inner loop starts that processes input from each field on the screen, one field at a time. This is accomplished in the subroutine at line 2000, and here's where things get tricky:

```
2000   field=fieldnum
2002   value$=""
2005   IF input.req%(field)=0 THEN RETURN
2006   IF input.req%(field)<0 THEN value$=name$(field,1):RETURN
```

ROLL UP YOUR SLEEVES

These first few lines are fairly obvious. Using the input.req% array,we can quickly determine if the field is one requiring only default processing or none at all. Note that the string Value$ will be used to convey the result of this field's

data entry process. Once we determine that actual input must take place, then the real work begins, as shown below:

```
 2008   row=info%(field,0)
 2010   start.window=info%(field,1)+LEN(name$(field,0))-1
 2015   field.len=ABS(info%(field,2))
 2020   IF field.len<>0 THEN end.window=start.window+field.len-1:GOTO
2060
 2021   GOTO 2060
 2035   IF info%(test,0)=row THEN end.window=info%(test,1)-1
 2036   field.len=end.window-start.window+1:GOTO 2060
 2040   test=field+1
 2045   GOTO 2030
 2060   WINDOW start.window,row TO end.window,row
```

This routine sets up a field for data entry. Because of the console driver's powerful windowing capabilty, once the size of the field is determined it is possible to construct a cell on the screen for each data item that must be input. As you can see, this window definition is relatively easy if the field length is known up front. Lines 2025 through 2045 are designed to determine the actual field length available to a variable-length item, by looking ahead at what's next on the screen and adjusting accordingly. Once that is determined, line 2060 establishes the window in which data entry takes place for that item. Next is the printing of any default values and the display (in inverse) of the field available for entry:

```
 2065   line$=MID$(blank$,1,field.len)
 2070   default$=name$(field,1):IF default$<>""THEN SUB$(line$,1,LEN
(default$))=default$
 2075   INVERSE:HOME
 2080   PRINT line$
 2082   PRINT set.edit$;
 2085   HPOS=1:point=1
```

Of note here is the variable Set.edit, which is used to turn off all console options, leaving the progam totally in control of cursor movement, wrap, scroll and so on. Line 2085 then positions the cursor to the beginning of the field (remember that this is a window now) and sets up a pointer "point" to the first character of the field value (Line$). Now the fun really begins:

```
 2100   ON KBD GOTO 2200
 2105   NORMAL:PRINT MID$(line$,point,1);:INVERSE:FOR j=1 TO
150:NEXT:PRINT MID$(line$,point,1):FOR j=1 TO 150:NEXT:GOTO 2105
```

This is our old friend the on kbd loop. In this case, we are using the normal and inverse options (and the fact that we just turned the console "advance after printing" function off) to blink whatever character in Line string we are curently pointing at. For the purposes of this routine, you can equate Line$ with what's

seen in the window exactly. Of course, you hum around in the little loop in line 2105 until a key is pressed. That sends the program off to line 2200:

```
 2200   OFF KBD
 2205   IF KBD<32 OR KBD>127 THEN 2270
 2210   SUB$(line$,point,1)=CHR$(KBD)
 2215   INVERSE:PRINT MID$(line$,point,1);
 2220   IF point<field.len THEN point=point+1
 2250   HPOS=point
 2255   ON KBD GOTO 2200
 2260   RETURN
```

FUNNY CHARACTERS

After checking for control or special function characters in line 2205, the typed character is inserted into the Line$ string at the current cursor position, and the character is reprinted in inverse to be sure that the on kbd routine wasn't exited in the wrong state. Assuming that there is room in the the window, lines 2220 and 2250 update the pointer and advance the cursor to the new position. Then lines 2255 and 2260 clean up and return to the blink routine, awaiting another keystroke. But what about those special characters? Wait no longer:

```
 2270   IF KBD=27 THEN escapecode=1:POP:RETURN
 2275   IF KBD=8 AND point>1 THEN INVERSE:PRINT MID$(line$,point,1);:
point=point-1:GOTO 2330
 2280   IF KBD=21 AND point<field.len THEN INVERSE:PRINT MID$
(line$,point,1);:point=point+1:GOTO 2330
```

These lines check for escape and exit back to the calling level (the pop gets us out of the on kbd routine and back to reality). In addition, lines 2275 and 2280 process the cursor keys for left and right arrow, first reprinting the current character and then resetting the pointer.

```
 2300   IF KBD=13 THEN value$=MID$(line$,1,point-1)
:line$=MID$(value$,1,field.len):GOTO 2320
 2305   IF KBD<>141 AND KBD<>9 THEN 2350
 2310   value$=line$
 2320   NORMAL:HOME:PRINT set.normal$;line$;
 2325   POP:RETURN
 2330   HPOS=point
 2350   ON KBD GOTO 2200
 2355   RETURN
```

These lines wrap up the routine once the user is satisfied that the field is complete. There are several options to signal completion. First, line 2300 processes the return key, discarding anything to the right of where the return key was pressed. Value$ is set to what's left and Line$ is redefined so that the actual data can be displayed in line 2320. Line 2305 processes the other option,

full entry of whatever is in the window, no matter where the cursor is. As you can see, this occurs when either open-apple return or tab is pressed. Set.normal$ turns advance back on, so that the value can be printed back into the window, this time with inverse off, to indicate that data entry is finished in that field.

WHEW!

All of that excitement leads us back to the main loop, now at line 120, where the result of the field call is analyzed. If escape was pressed, a further check is made to see if it was pressed in the first input field of the form. If so, that is the indication to terminate input of forms, and the program jumps out of the loop. If escape is pressed in any other field, processing starts over at line 105 with a clean slate. If the return was normal, with data for the field in Value$, then there is an opportunity to do any additional processing required and then add Value$ to the accumulating Out. rec$ in the routine at line 3000. After all the fields are processed, line 215 and 220 display it,and if the file logging option was set originally, the subroutine at line 4000 writes the result in a file. Here are simple examples of what these routines look like:

```
3000 IF LEN(value$) THEN out.rec$=out.rec$+value$
3005 RETURN
4000 PRINT#2,recordnum;out.rec$
4010 RETURN
```

Obviously, "real" data entry programs will have much more elaborate processing and editing functions built in. This example was only a guide to how you might incorporate these techniques into your own programs.

TRY THIS ONE

Some things you might want to try in order to improve the program could include expanding the info% array to contain more information about editing. (Such as: If the data alphabetic or numeric? Does it have a fixed decimal place? Can it have a null value, or must some nonblank or non zero value be used?) For fixed record layout output (like simulating records on a keypunch machine-yuck!) you might want to add fields to define where in the output record the value is to be placed (starting byte and length, for example). If you are really clever, you can modify the routine to accept multiline fields. Remember, also, that there is nothing sacred about the beginning of the program, either. The subroutines could just as easily be used within a completely different environment to support your program's screen-handling needs.

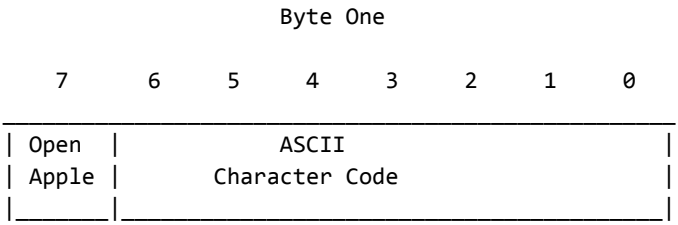
FINAL LAST CHALLENGE (MAYBE)

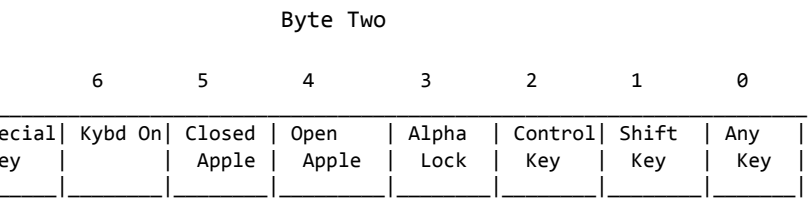
Last month's Last Challenge wasn't. It actually applies to this month's Third Basic, so look back and have fun.

Exploring Business Basic - Part XIII

Last time we plunged as far as anyone seemed to dare into creating special screen functions. As is the usual pace in this column, we will now move swiftly from the sublime to the overly intense, and in the process answer last month's mystery question, "How can I tell the difference between RETURN and ENTER?". This question, or one at least similar to it, was asked last time when we delved into creating a data entry screen builder program. Rather than answer the question by inserting that specific capability into the previous program, the following is a general purpose keyboard read program which you can modify to any number of uses.

Before getting into the program, let's consider some possible solutions and their problems. We have agreed that the only way to tell the difference is to use the fact that the ENTER key, along with the rest of the numeric pad and some other keys, is a "special" key within the SOS keyboard definition. That means that although a normal read to the console returns an ASCII 13 in both cases, a "two-byte" read will return a flag in the keyboard status byte that indicates whether or not a special key was pressed. The layout of the two bytes looks like this (stolen from Appendix G of the Standard Device Drivers manual):





Note: "Keyboard On" and "Any Key" will normally be "1" for any keypress, the other bits will be "1" only if that particular function is active.

Well, this looks deceptively simple. We learned in a previous episode that there is a control call to the .CONSOLE driver (using REQUEST.INV) which can put the keyboard into two byte read mode. This is device control call number 3. After calling it with a parameter of 128 (hex 80) the console will return two bytes for each keypress. OK! Now all we have to do is perform the control call and input

from the console, right? Unfortunately, ordinary reads don't work too well. They keep expecting the line terminator character (normally RETURN) and when two bytes keep popping up, INPUT gets confused. Ok, still simple, let's do a GET, which doesn't require a terminator character. Oops again. GET expects to receive only one character, and in fact informs the console of this desire. The console keeps getting two characters each keypress, and for reasons too bizarre to discuss here, returns no characters to the GET. GET, expecting always to receive a character as soon as one is typed, bravely returns a null string to the user, no matter what was typed. Interestingly enough, the ASC function interprets a null string as having an ASCII value of -1 (You may have to try that one to believe it).

Now that all that's clear, lets get back to the original question. How can you do it? The most plausible answer lies in yet another console capability, the little known "no-wait read". We want to read all the characters in the input buffer, anytime, without a termination character. The console control call number 10 with a parameter of 128 does just that. With that, lets look at a program to monitor the keyboard and print out the two-byte code for whatever is typed.

```
 5 INVOKE"request.inv"
 10 HOME
 15 PRINT"Test two byte reads"
 20 PRINT:PRINT"Input buffer contains: ":HPOS=0:VPOS= VPOS-2
 25 OPEN#1,".console"
 30 hex80$=CHR$(128):hex00$=CHR$(0):clearendvp$=CHR$(29)
 35 device$=".console"
 40 PERFORM control(%3,@hex80$)device$
 45 PERFORM control(%10,@hex80$)device$
 50 ON ERR GOTO 100
 55 ON KBD GOTO 65
 60 PRINT"/";:FOR j=1 TO 200:NEXT j:HPOS=HPOS-1:PRINT"\";:FOR j=1 TO
200:NEXT j:HPOS= HPOS-1:GOTO 60
```

Line 5 invokes the REQUEST.INV module, which performs SOS calls. Then lines 10 through 35 set up the screen and initialize variables which will be used later. Line 40 performs the control call which puts the console into two-byte read mode. Line 45 puts the console into no-wait mode, so that read requests are immediately filled with whatever has been typed up to the point of the read. Line 50 is VERY IMPORTANT. It sets up a jump to a "back to normal" routine. If for some reason your program terminates without setting everything back, BASIC will be VERY confused, and you will have to reboot. This is because BASIC, like everybody else, uses the console driver for input, even to a "prompt".

Line 55 starts the interesting stuff. We could, of course, keep reading the console until something showed up (sometimes called "polling" the keyboard). A

better, more efficient way is to use the keyboard interrupt as a trigger to go and look at what was typed. Between keypresses, line 60 keeps something interesting going on the screen, to let you know that it's waiting. Obviously, the routine at line 60 could be expanded to do useful work (more on that later!).

When a key is pressed, the ON KBD statement in 55 sends the program leaping to line 65. That routine looks like this:

```
 65 OFF KBD
 70 INPUT#1;char$
 75 PRINT:HPOS=24
 77 FOR k=1 TO LEN(char$):PRINT" ";MID$(HEX$(ASC(MID(char$,k,1))),3,2);:
NEXT k
 79 PRINT clearendvp$:PRINT
 80 IF char$=CHR$(9)+CHR$(199) THEN 100
 85 HPOS=0:VPOS= VPOS-3
 90 ON KBD GOTO 65
 95 RETURN
```

First, we turn off keyboard interrupts, and then perform a file INPUT statement to get the accumulated characters. Using file INPUT (INPUT#) is unusual, since it is generally easier to use the value of KBD, the reserved value containing the ASCII value of the keypress that triggered ON KBD. However, if you think that a two-byte read confuses GET, it really blows KBD's mind. Note that an ordinary INPUT statement with no-wait on only returns the first byte in the buffer (don't ask me why!). Thus we use INPUT#, and since no-wait is turned on, we'll get any characters currently in the buffer placed in the variable "char$".

Line 77 scans through the string, and prints out the HEX codes of each character there. By scanning the entire length of the string, we take care of the case where rapid typing managed to enter characters while the previous set of characters were being processed. Don't let the somewhat complex print statement in line 77 throw you. It is simply taking each character, converting it to its ASCII equivalent, converting that to hexadecimal notation, and then extracting the rightmost two Hex digits (since the value is always 255 or less - FF in hex). Line 79 prints the console command to "clear to end of viewport" (CHR$(29)). This insures that if the previous display contained multiple characters, the excess ones will be erased when the PRINT occurs.

Line 80 gives us a way out of the program, by testing for a certain two-byte combination. Checking your keyboard chart should prove that the desired combination (read as an ASCII 9 followed by an ASCII 199) is a "control-shift TAB". You can change the exit to any combination of keystrokes just by substituting the appropriate character codes in line 80. If the characters don't match, the routine resets the display location and returns to the little time-waster in line 60.

In the event of a match (or an error), line 100 through 125 clean things up and terminate the program. They look like this:

```
100 REM return to reality
105 PERFORM control(%3,@hex00$)device$
110 PERFORM control(%10,@hex00$)device$
115 PRINT:PRINT
120 CLOSE:INVOKE
125 END
```

Now that you've typed in this little jewel, you can try some interesting things. So far I've discovered ten different variations on the letter "a", and there are bound to be more. For those who think the Apple III only has two function keys, think again! The combinations are practically endless.

When you are tired of trying out all the weird combinations (like Open-Apple, Closed-Apple, Control, Shift, Alpha Lock A), then consider the more useful ones. RETURN is read as "0D 41" which means carriage return with "keyboard on" and "any key" flags set. ENTER, on the other hand, is read as "0D C1" which means carriage return with "keyboard on", "any key" and "special key" flags set. Similarly, a "1" on the main keyboard is read as "31 41", while "1" on the numeric pad is "31 C1". That's how programs like "Word Juggler" can use the numeric pad as a special function key set. Another handy example is "control-H" is a "08 45" while the backarrow key is "08 C1". This allows a program to distinguish between an ASCII backspace, and a cursor backspace. Applewriter III uses "control-backarrow" (a "08 C5") for deleting a character and a simple backarrow ("08 C1") for cursor movement. With single byte reads, these two different combinations would be indistinguishable.

As usual, this column tells you more than you could possibly want to know about almost everything. In the case of two-byte reads, the result of knowing all this stuff is the possibility of developing some really friendly applications which use the keypad and other keystroke combinations to really simplify things for the user. An added benefit, and possibly the subject of a full article one of these days, it the ability to do useful work within an application while waiting for the user to type on the keyboard. One piece of useful work would be to print out a disk file which was previously written by the application. You could use the driver status calls which tell how many characters are left to be printed by a printer driver and occasionally print enough to keep the printer busy. In some circles, this is known as "spooling" and is considered really tricky. With SOS and the input routine above, it becomes relatively trivial. Good luck, and have fun!

P.S.: Next month's column is a long treatise on sorting techniques in Basic, including some routines that can sort a thousand items in a minute or so. Unbelievable? Watch this space! Until then, a final puzzle. What combination

of keys creates the largest value for a two-byte read (considering both bytes as one 16 bit unsigned value)? Answer next time!

Exploring Business Basic, Part XIV

Welcome back to Basicland. Before we plunge waist-deep into today's exciting episode, it's time to congratulate Arnold Bailey of New York State for his intrepid solution to our challenge on two-byte reads from the console. For those who missed the last two cliff-hangers, we wrapped up the discussion of nifty ".console" features by challenging you, the studio audience, to come up with a routine in Business Basic that allowed a program to tell the difference between the "Enter" and "Return" keys. Last month a solution was furnished in this column, but because of publishing lead times, it had to be submitted before the challenge itself could be issued (maybe a better name for the column is Tommorrowland!) The published solution relied heavily on the desire to read a keystroke at a time and sample each one, simulating the "GET" statement in BASIC. Arnold took advantage of the fact that the console still terminates a read on an ASCII 13 (which Return and Enter both generate) and correctly identified that an INPUT# statement was required to correctly read both bytes. It was an excellent solution, and well documented (embarrassingly well documented compared to the abbreviated listings normally foisted upon you in this column). For being first with a valid solution, Arnold wins a copy of "Quickfile III" and the thanks of a grateful nation.

Sorting it all out

One of the key ingredients of most business and scientific programs which handle large amounts of data is the ability to arrange that data in an ordered sequence. "Arrange data in an ordered sequence" is, of course, a windy way of saying "sorting". There are as many sort techniques as there are people to think them up, but for the purposes of this column and its Christmas (December) cousin, we will stick to four or five fundamental methods. Business Basic has several nifty features which make sorting more efficient, and the huge memory space makes sorting large collections of strings or numbers practical to do in memory. For that reason, "Merging", the flip side of most sort techniques, will only be briefly covered in the December issue. For now we'll stick to in-memory sorts to illustrate the techniques.

I'm forever showing bubbles

The most common sorting technique, and the one guaranteed to show up in every elementary textbook, is the "bubble" sort. So named because of the technique of taking a value and shuffling (bubbling) it up to its proper place in the list, it depends on comparing each value to the one next door, and exchanging them if the order is wrong. If you compare each element with its neighbor enough times, eventually the list will be sorted. For the purposes of

the sample program, and all the rest of the programs in this series, we will assume the desire is to sort the lists in "ascending" (lowest to highest) order. Let's plunge into the first example to illustrate how this works:

```
 10   REM sort using bubble technique
 20   REM   n is number of elements
 30   REM   sarray is the array to be sorted
 50   REM
 100   DIM sarray(1000)
 110   INPUT"Sort Routine.  Number of elements to generate: ";a$
 120   n=CONV(a$):IF n<2 THEN 200
 130   FOR i=1 TO n:sarray(i)=RND(1):NEXT
 135   PRINT"Start of Sort"
 140   GOSUB 1000
 150   PRINT"Sort complete. First 10 elements are:"
 160   FOR i=0 TO 10:PRINT sarray(i);" ";:NEXT
 170   PRINT:PRINT:GOTO 110
 200   END
 1000   top=n-1
 1010   madeswap=0
 1020   FOR i=1 TO top
 1030     IF sarray(i)>sarray(i+1) THEN SWAP
sarray(i),sarray(i+1):madeswap=1
 1140     NEXT i
 1050   IF madeswap THEN top=top-1:GOTO 1010
 1071   RETURN
```

This first program sorts a numeric array, rearranging it in memory. For purposes of testing it, lines 110 through 140 ask for the number of elements to sort, load "sarray" with random values, and then call the subroutine at line 1000 to perform the actual sort. After returning from the sort, lines 150 through 200 print out the first ten elements (just to demonstrate that the data is sorted) and end. Remember that this framework is deliberately simple, so we can concentrate on the sort technique itself.

Line 1000 establishes the upper limit of our check for correct order and line 1010 establishes a variable "madeswap" which is a flag we'll use later to determine if more sorting needs to be done. That leads us the the main routine in lines 1120 to line 1040. Line 1030 scans each element and compares it to the next higher element. If the first element is greater than the second, then the SWAP statement is used to exchange them, and the "madeswap" flag is set to show that an exchange was made. Then the process repeats with that next higher element compared with its next highest companion, until the top is reached. Note that's why "top" is set equal to the number of elements minus one. When a complete scan is made, line 1050 checks to see if any swaps were made in the last pass. If there were not, then the array must be in order. If not,

the array might not be in order, and needs to be processed again until all possible swaps have been made. Notice that line 1050 resets "top", the limit of checking, since after each pass the largest number found anywhere in the array will be forced to the top of the list (try it out if you don't believe it). This is something that some versions of this routine miss, and it leads to a lot of unnecessary scanning. Notice also that we take advantage of the SWAP statement. Some Basics don't have this statement, and the result is that you have to assign one value to a temporary variable, do the reassignments, etc. This also costs a lot of time, whereas SWAP is very fast. As long as things are being noted, its probably worth pointing out that this routine could be rewritten for descending order by rearranging the sense of the IF statement and having the search go from a varying "bottom" to a fixed "top" instead of the other way around. One last comment. Although this sort technique is the simplest, it is also the slowest, except for those situations where the list is very short or almost sorted already. Simple timing tests will convince you that the routine slows down non-linearly as the size increases. "Non-linearly" means it goes from "bad" to "awful" without passing through "worse".

Getting the point

The routine below is another variation on the bubble theme, with one important exception. Sometimes it doesn't make sense to actually rearrange the data, but rather only to create a list that describes what the order would be if they were physically sorted. For example, consider the following list (as it might have been read from a file):

```
Record number             Item

      1                   Henry
      2                   Bill
      3                   Gloria
      4                   Alphonse
      5                   Gaston
```

One way to sort this list is to simply arrange it in the following sequence:

```
Alphonse, Bill, Gaston, Gloria, Henry
```

That's ascending alphabetical order. However, we could represent that same sequence by listing the "record numbers":

```
  4  2  5  3  1
```

Not only is this second representation more compact, it may actually represent less work to rearrange the "record numbers" than the actual data itself. Such numbers are called "pointers", since the value that is listed is just a pointer to the actual data location, not the data itself. If you want to construct the sorted

list of names, it is easy to look them up using the record number (pointer) list. Some languages and systems make a great deal out of this pointer concept. For now, the suggestion should be to use the technique wherever it makes sense from a performance, storage and convenience standpoint. The example below is an adaptation of the first program to incorporate this "pointer sort" technique:

```
 10   REM sort using bubble technique
 20   REM   N is number of elements
 30   REM   sarray is the array to be sorted
 40   REM   parray is the pointers to the sorted array
 50   REM
 100   DIM sarray(1000),parray%(1000)
 110   INPUT"Sort Routine.  Number of elements to generate: ";a$
 120   n=CONV(a$):IF n<2 THEN 200
 130   FOR i=1 TO n:sarray(i)=RND(1):parray%(i)=i:NEXT
 135   PRINT"Start of Sort"
 140   GOSUB 1000
 150   PRINT"Sort complete. First 10 elements are:"
 160   FOR i=1 TO 10:PRINT sarray(parray%(i));" ";:NEXT
 170   PRINT:PRINT:GOTO 110
 200   END
 1000   top=n-1
 1010   madeswap=0
 1020   FOR i=1 TO top
 1030     IF sarray(parray%(i))>sarray(parray%(i+1)) THEN SWAP parray%
(i),parray%(i+1):madeswap=1
 1040     NEXT i
 1050   IF madeswap THEN top=top-1:GOTO 1010
 1060   RETURN
```

Notice that we have introduced a new array in the program at line 100. "Parray" contains the pointers to the actual locations in "sarray" which represent the sorted order. Notice in line 130 that "parray" is set initially to the sequence 1, 2, 3 ..., the same sequence as we find the data in "sarray" initially. However, since the pointer array contains only references to locations in the original array, it can be declared an integer array (maximum value 32767) to save space. Having set all the values up, the GOSUB to line 1000 sorts the data, as in the first example, except now in line 1030, instead of testing the actual sarray values directly, we use parray%(i) and parray%(i+1) as pointers to where the real data is. Once the comparisons are made, the pointers (not the values themselves) are physically exchanged. When the sort is finished, the routine returns to line 150 to print out the sorted data. The PRINT statement in line 160 illustrates how the pointer array is used to look up the correct sequence of values, even though they are actually scattered around within "sarray".

One of the interesting possibilities of this technique is that it is possible to have more that one pointer array to a given data array. In that way, you could have an "sarray" which had associated with it a "parrayup" and a "parraydown" pointer array, so that listings and searches could be done in either order, depending on the program requirements, without resorting. These are sometimes called indexes, and are very useful in many applications. More typically, pointer arrays are used in situations where the original array consists of string values, or fields within disk records. There the economy of storage of an integer array is very valuable, and the pointers consist of actual disk record numbers, which are then easy to look up in any specified order. Next time we will consider some sort techniques and pointer arrays which are particularly suited to disk lookups. These techniques are sometimes generalized under the heading "access methods".

A Mild Speed Lift

All this is fine, but the original warning still is worth considering. Bubble sorts are simple and easy to understand, but they are painfully slow on any reasonable amount of data. The fundamental problem is that the algorithm requires lots of comparisons, and even if the comparison indicates that the data needs to be moved, a move of one cell at a time is all that is possible. That means that for a small value to get from the top to the bottom on an ascending sort requires lots of exchanges (one for each value in the array). One quasi-obvious way to speed this process up is to make comparisons across larger distances, and thereby cut down the number of compares and exchanges necessary. A sorting algorithm called the Shell-Metzner sort accomplishes this. Usually called the "Shell Sort" (which is appropriate considering its similarity to the "shell game" switching technique), it depends on long distance comparisons and swaps to speed up the sorting process. For simplicity, we'll look at the Shell sort in standard form, without the additions for pointer sorting. A typical Shell sort routine looks like this:

```
10   REM sort using Shell-Metzner technique
20   REM   n is number of elements
30   REM   sarray is the array to be sorted
50   REM
100   DIM sarray(1000)
110   INPUT"Sort Routine.  Number of elements to generate: ";a$
120   n=CONV(a$):IF n<2 THEN 200
130   FOR i=1 TO n:sarray(i)=RND(1):NEXT
135   PRINT"Start of Sort"
140   GOSUB 1000
150   PRINT"Sort complete. First 10 elements are:"
```

```
 160   FOR i=1 TO 10:PRINT sarray(i);" ";:NEXT
 170   PRINT:PRINT:GOTO 110
 200   END
 1000   IF n<2 THEN RETURN
 1010   span=INT(n/2)
 1030   newspan=n-span
 1040   FOR i=1 TO newspan
 1050     temp=i
 1060     upper=temp+span
 1070     IF sarray(temp)>sarray(upper) THEN SWAP
sarray(temp),sarray(upper):temp=temp-span:IF temp>0 THEN 1060
 1080     NEXT i
 1090   span=INT(span/2)
 1100   IF span THEN GOTO 1030:ELSE RETURN
```

As you can see, the initial routine from line 10 to 200 is essentially the same as in the other routines. The subroutine in line 1000 implements the Shell algorithm, starting with a quick check in line 1000 that there is more than one element to be sorted. Once that is established, line 1010 divides the array into halves, and line 1030 established the center point around which swaps will be made. It would be a good idea to create a small array on paper and follow through on exactly how this routine works for your own satisfaction. Briefly, the "newspan" variable establishes a pivot point with the loop in lines 1040 through 1080 facilitating conparisons and swaps between the upper and lower portions of this pivot point. After each swap, the range of search is narrowed by decreasing the "temp" value in line 1070, and the process is repeated. After each major pass with a span value, line 1090 cuts the span value in half and repeats the process, until the span is one, at which point the array is sorted. Line 1100 checks for that happy occurrence, and returns if it is so.

Several things are worthy of note in this routine. First, the Shell sort will work faster on sorted data than unsorted data, and in nearly every unsorted case, will out-perform the bubble sort, dramatically so in cases above fifty to one hundred values. Also, this routine can easily be adapted to a pointer sort, using the techniques outlined in the second example above. String arrays can be sorted simply by replacing the numeric comparison in line 1070 with a string array compare. The SWAP statement works equally well with string and numeric data.

Sort of a new way to sort

The algorithms above are fairly standard and safe, and will reliably sort any kind of data. Sometimes in application programs we are more fortunate, and can have special knowledge of what kind of data we are sorting. This allows for special techniques which are faster than any "general purpose" routine could

be. Although the Apple III is one of the fastest personal computers around, its not exactly a Cray I, so lots of times it pays to be able to pull tricks like the one below. Imagine a situation where there are lots of records to sort, but there are only a few unique values among all the records. One classic example is sorting address records on the basis of the value of a "State" field. Obviously, there may be thousands of records to sort, but there are only 50 possible states, each typically represented as a two character code. There are many other examples, but that is one which is easy to imagine.

The program below generates random string values, and then lets you test the practicality of a sort technique based on a concept called an "inverted list". Inverted lists are a favorite topic around "access method" and "database" experts, but the same principles can apply to sorts. Basically, an inverted list is not an upside down version of a regular list, as you might expect from the title. Rather, you can think of it as a list of all the unique values in another list, with sublists which contain the record numbers of all the records sharing the same field value. In our example above, if we had the following situation:

```
Record number              State

      1                     CA
      2                     MD
      3                     CA
      4                     NY
      5                     MO
      6                     CA
      7                     MD
```

Then the inverted list of this data would look like this:

```
Value         Locations

 CA            1, 3, 6
 MD            2, 7
 MO            5
 NY            4
```

The two representations contain the same information, but in a considerably different form. Note also that the inverted list is assembled in ascending alphabetical order. That's not necessary to the example, but once the unique values are established, it is generally easy to sort them. This is especially true if the number of unique values is much smaller that the total number of values. The program below lets you generate random string arrays, with up to a thousand values, and then pick a group of columns on which to sort. Try it initially by sorting only on the first column. This will guarantee that there are only 26 unique values (since the routine generates only upper-case letters), no matter how many strings you generate. First, the main routine:

```
 10   DIM  pntr%(1000),startval%(255),endval%(255),spointer%(255),
sarray%(1000)
 20   zero$=CHR$(0):zero%=0
 30   DIM array$(1000),value$(128)
 40   INPUT"Number of strings to generate: ";a
 50   INPUT"Number of characters per string: ";b
 60   FOR i=1 TO a
 70     FOR j=1 TO b
 80       array$(i)=array$(i)+CHR$(INT(RND(1)*26)+65)
 90       NEXT j,i
 100   FOR i=1 TO a:PRINT array$(i);" ";:NEXT i
 130   PRINT:PRINT:INPUT"Start and end columns for sort: ";c1,c2
 132   FOR i=1 TO a:pntr%(i)=zero%:NEXT:FOR i=1 TO
255:startval%(i)=zero%:endval%(i)=zero%:NEXT
 133   sortval$=""
 140   FOR rec%=1 TO a
 150     item$=MID$(array$(rec%),c1,c2-c1+1)
 160     GOSUB 2000
 170     NEXT rec%
 190   GOSUB 3000
 196   PRINT"Number of unique values: ";n:PRINT"Ratio of total records
to unique values: ";a/n
 197   INPUT"Press return for the sorted list: ";a$
 200   FOR i=1 TO a:PRINT array$(sarray%(i));" ";:NEXT i
 210   PRINT:PRINT:INPUT"Sort the array again? ";a$
 220   a$=MID$(a$,1,1):IF a$="Y" OR a$="y" THEN 100
 230   END
```

Lines 10 and 20 set up a lot of values which will be used later, while line 30 sets up the main string array "array$" and the array which will contain unique values "value$". After prompting for the number of strings and the size of each string, lines 60 through 90 build the string array by randomly creating character strings composed of upper-case ASCII characters. Line 100 prints out the created array, and line 130 requests the columns on which to sort. Unless you create very few strings, it is best to sort on only one column, since the unique combinations possible in sets of more than one column are probably too great for the routine to work properly. Note that in a controlled (non-random) set, like the States, this might not be a problem. In any case, once the columns are chosen, line 132 and 133 initialize values and prepare for the main sort loop in lines 140 through 170. Note that for each record (rec%) to be sorted, the variable "item$" contains the value extracted from the main record which will become the sort key for that record. The routine at line 2000, which we will examine shortly, adds the current value to the list of unique elements if necesary, and inserts its record number in the general list. The subroutine at line 3000 then orders the unique value list, creates the sorted pointer list in "sarray%" and

returns to lines 196 through 220 to print the list on demand and start the process over, if desired.

Lets look now at the routine that creates and adds to the inverted list:

```
 2000   x=INSTR(sortval$,item$)
 2010   IF x THEN pntr%(endval%(x))=rec%:endval%(x)=rec%:RETURN
 2020   x=LEN(sortval$)+2
 2025   IF x+LEN(item$)>255 THEN PRINT"sortval$ overflow, sort
aborted":STOP
 2030   sortval$=sortval$+zero$+item$
 2040   startval%(x)=rec%
 2050   endval%(x)=rec%
 2060   RETURN
```

This routine makes good use of the INSTR function, to search a string called "sortval$". Sortval$ contains all the unique values, separated by the ASCII value 0. This means that the unique values can easily be identified, assuming that none of the values contain an ASCII 0 themselves. Once INSTR either finds or doesn't find the "item$" value in "sortval$" the rest of the routine is set into motion. In the case that "item$" already exists in "sortval$", line 2010 updates the "pntr%" array by putting the current record number into the space reserved for the last record number in the list of that unique value. That is, "endval%" is an array which remembers the index in "pntr%" of the last occurence of any particular unique value. That means that the next occurence of that value gets automatically put into the location in "endval%" that matches the start location of the value in "sortval$". This is another good one to try on paper until you get a feel for how it works. Assuming that the value was found, the routine's work is finished for now, and it returns to look at the next value.

If the item was not found in "sortval$", then that means that is is a new unique value. Line 2020 gets the next possible location for storing the new value, and line 2025 checks to see if there is room for the value. You could probably come up with something more friendly than the "STOP" statement to solve the problem. In any case, if there is room, line 2030 adds the value to the "sortval$" string, with the "zero$" spacer, and 2040 and 2050 establish start and end locations for this new value and then return to get the next record.

This process continues until all the records are examined and all unique values are added to "sortval$" and their beginning and ending pointers are established in the appropriate arrays. At this point "pntr%" contains a linked list for each unique value of "sortval$", with the starting point of the list pointed to by the appropriate element of "startval%" and the end point defined by a zero in the location pointed to by "endval%". Now the fun begins. Having assembled the list of pointers to all values, it is necessary to sort the unique values themselves into the appropriate order, and then assemble the individual linked lists into a

total sorted list. The routine to break out the unique values and sort them looks like this:

```
 3000   sortval$=MID$(sortval$,2)
 3002   FOR n=1 TO 255
 3005     x=INSTR(sortval$,zero$)
 3010     IF x=0 THEN 3050
 3015     value$(n)=LEFT$(sortval$,x-1)
 3020     sortval$=MID$(sortval$,x+1)
 3025     NEXT n
 3050   value$(n)=sortval$
 3055   last=1
 3060   FOR i=1 TO n
 3070     FOR j=last TO 255
 3080       IF startval%(j)<>0 THEN spointer%(i)=startval%(j):GOTO 3100
 3090       NEXT j
 3095     PRINT"ERROR, startval not found":STOP
 3100     last=j+1
 3110     NEXT i
 3120   GOSUB 4000
```

Lines 3000 through 3050 scan the "sortval$" array and break out each value into a separate element of "value$" for ease of lookup and sorting later. It relies on "zero$" as a delimiter between values in "sortval$". One note here may help understanding. The whole reason why "sortval$" was used instead of going with a string array for the values was because INSTR is an infinitely (nearly) faster way of searching for a given string value than a FOR-NEXT loop plowing through "value$", and since that operation has to be done for each record, speeding up the search was a critical issue. In any case, once "value$" is built, a corresponding list of the start values for each string is built in "spointer%" by lines 3055 through 3110. This leaves us with a list of the actual values, and the beginning values of the linked list for each. Now all that remains is to sort the values themselves, and rearrange the "spointer%" list to match. That's done in the subroutine at line 4000:

```
 4000   IF n<2 THEN RETURN
 4010   span=INT(n/2)
 4020   newspan=n-span
 4031   FOR i=1 TO newspan
 4040     temp=i
 4050     upper=temp+span
 4060     IF value$(temp)>value$(upper) THEN SWAP
value$(temp),value$(upper):SWAP spointer%(temp),spointer%(upper)
:temp=temp-span:IF temp>0 THEN 4050
 4070     NEXT i
 4080   span=INT(span/2)
```

```
 4090   IF span THEN GOTO 4020:ELSE:RETURN
```

That's right, campers, its our old friend (of a page ago) the Shell sort. The only change is that we are sorting string data, and in addition to swapping the string values, we swap the "spointer%" values as well. The important thing here is that even though we may have processed thousands of records, we only have to sort the unique values among them. As long as we have lots more records than unique values, this routine will save significant amounts of time. Anyway, to finish up, once the values are sorted, we can assemble the whole list of record pointers by following the start values in the "spointer%" array, and loading all the elements of the linked lists in the new order. That will finish the subroutine, and looks like this:

```
 3130   k=0
 3140   FOR i=1 TO n
 3150     index=spointer%(i)
 3160     k=k+1:sarray%(k)=index:IF pntr%(index)<>0 THEN
index=pntr%(index):GOTO 3160
 3190     NEXT i
 3200   RETURN
```

Notice that each linked list starts with an index in "spointer%" and ends when the value in "pntr%" is zero. By assembling the lists one by one in the sorted sequence determined by "spointer%" we guarantee that the whole list is in order.

Glancing way back up to the original main program, you can see that line 196 through 220 can now take the "sarray%" list as a pointer list into the original records and print out that list in order.

It's hard to believe that the original "bubble sort" program in the beginning of this tome can be so short and so simple and yet take the longest to execute, while the last program, which seems so complicated and so long can do certain types of sorts at least a hundred times faster. Many times it's not how much code is in the program, but how many times it must be executed which really makes the performance difference. For that reason, in sorting as well as any other activity, it really pays to examine your loops and repetitive code, and to think of the best algorithms possible. Remember too, that the last program is useful only if there are only a few unique values, and the safest bet in the general case is the Shell sort.

Next time we will take up some other interesting sort techniques, including an improved Shell sort called the "Quick sort" and a completely different sort called the "Binary tree sort". The Binary Sort, like the inverted list sort discussed in this article, can also be the basis for an access method. In fact, Apple III's Record Processing Services package uses a modified version of this algorithm.

Hopefully, we'll get the chance to get into those techniques as well. Until then, don't get "out of sorts"!

Exploring Business Basic, Part XV

Lots of exciting things have happened on the Apple III frontier since last we talked, and if any of you haven't heard about the new products Apple announced at Comdex, you should really check them out at your dealer's. One other product is worthy of note as well, a new piece of software from Quark Engineering (the Word Juggler people). The package is called "Catalyst", a perfect name, since it can control the startup and execution of all your Apple III programs from a single command menu. This means that you can load all your application programs and language interpreters onto mass storage devices (like Profile!) and once the Catalyst program is booted, going from Applewriter to Basic to Visicalc to Senior Analyst to Pascal to System Utilities is as easy as hitting a special keystroke and picking the application off the menu. Yes, even protected programs like Visicalc are provided for! If you haven't seen this program yet, zip down to your dealer for a demo or get in touch with Quark, it's really something. One more note: The version of the console driver on the Catalyst disk implements several new events and allows the normal "read with wait" request to the console to be interuptable. This means things like generalized device spooling and some limited task switching are now possible for clever programmers. Rack up another first for Apple III and SOS!

Sifting through the Sorts

Last month we started exploring sort techniques, covering the simple but slow "bubble" sort, the Shell sort, and a somewhat esoteric sort that was referred to as an "inverted list" sort, good for situations where there were a large number of items but only a few values. Although the inverted sort is generally used for string data like state codes, sex codes, etc., it is also very useful on numeric data where the same principle (lots of items, only a few unique values) applies. This month we will look at a faster version of the Shell Sort, called the Quicksort, and a new sort technique called the "Binary Tree Sort". Let's look quickly at the Quicksort (heh,heh), and then explore the Binary sort in some detail.

The Big Shuffle

For simplicity, we'll start with the routine used last time to generate random numbers to sort in an array called "sarray". The actual sort routine will be contained in a subroutine. The main program looks like this:

```
100   DIM sarray(1000),stack(100)
110   INPUT"Sort Routine.  Number of elements to generate: ";a$
120   n=CONV(a$):IF n<2 THEN 200
130   FOR i=1 TO n:sarray(i)=RND(1):NEXT
```

```
135   PRINT"Start of Sort"
140   GOSUB 1000
150   PRINT"Sort complete. First 10 elements are:"
160   FOR i=1 TO 10:PRINT sarray(i);" ";:NEXT
170   PRINT:PRINT:GOTO 110
200   END
```

As you can see, line 130 generates a set of random values in "sarray" and then a GOSUB is performed to sort the array. As we discussed last time, one way to improve the performance of "exchange" type sorts (the name which the bubble, shell and quick sorts share) is to make the exchanges cover as much territory as possible in one swap. Quicksort algorithms go this one better by first picking a value which is a guess as the the midpoint of the values in the set, and then swapping other values based on this hypothetical middle value. Furthermore, the initial swaps are performed on the opposite ends of the array. The quicksort gets further speed by successively partitioning the sets of values into smaller groups and working on each until it is small enough to be sorted by simple swaps. While this makes for a much more complicated group of instructions, far less iterations of the code have to be performed, and thus faster performance is possible. Here's the routine:

```
1000   REM Initialize begin and end points
1005   l=1:stack(1)=n+1:m=1
1010   j=stack(l):i=m-1
1020   IF j-m<3 THEN 1100
1030   mid=INT((i+j)/2)
1040   i=i+1:IF i=j THEN 1060:ELSE:IF sarray(i)<=sarray(mid) THEN 1040
1050   j=j-1:IF i<>j THEN IF sarray(j)<sarray(mid) THEN SWAP sarray(i),
       sarray(j):GOTO 1040:ELSE:GOTO 1050
1060   IF i>=mid THEN i=i-1
1070   IF j<>mid THEN SWAP sarray(i),sarray(mid)
1080   l=l+1:stack(l)=i:GOTO 1010
1090   REM  check for cases of 1 or 2 elements
1100   IF j-m<2 THEN 1130
1110   IF sarray(m)>=sarray(m+1) THEN SWAP sarray(m),sarray(m+1)
1120   REM  Set begin and end points and check for completion
1130   m=stack(l)+1:l=l-1:IF l>0 THEN 1010
1140   RETURN
```

Several things are worthy of note here. First, it is necessary to use the same code to operate on each of the partitions of data which will be sorted. In some programming languages, this would be handled with a technique called "recursion". Recursion simply means that a routine can call itself, without limit. Basic not only has limits on how many times a GOSUB can be executed without a RETURN, it also has the attribute that all variables are global, in other words, each occurance of a subroutine will use the same variables over again, forgetting their previous state. For those reasons, and others, the routine above

uses the "stack" array to maintain information about each partition of the array, and to work its way through each partitioned set until all are sorted. Note also the extensive use of the SWAP command, to exchange values in the most efficient manner possible.

The quicksort algorithm is generally faster than the shell sort, except in those circumstances where the data is already sorted, or nearly so. Shell sorts will work through such arrays faster than the quicksort, which takes almost as long to arrange sorted data as to arrange data which is in random order.

One last example of quicksort would be appropriate. The example above used the direct sorting of numeric data as an example. Quicksort can be used just as easily to sort string data, or simply create sorted pointers to any type of data array. The example below takes string data from a file and performs a pointer sort. Remember that is is generally more efficient to swap numeric pointers than to swap strings. Voila:

```
100   DIM sarray(1000),stack(100),sort$(1000)
110   INPUT"Sort Routine.  Filename to sort: ";a$
120   IF a$="" THEN GOTO 200:ELSE:OPEN#1 AS INPUT,a$
125   ON EOF#1 LET sarray(0)=i-1:n=sarray(0):GOTO 140
130   PRINT TIME$;"   Start of sort"
135   FOR i=1 TO 1000:sarray(i)=i:INPUT#1,i;sort$(i):NEXT
140   GOSUB 1000
145   PRINT TIME$;"   End of sort"
150   INPUT"Sort complete. Do you want to list the sorted records?";a$
160   IF a$<>"Y" AND a$<>"y" THEN 110
170   FOR i=1 TO sarray(0):INPUT#1,sarray(i);a$:PRINT a$:NEXT
180   GOTO 110
200   END
```

The main routine has been changed considerably. "Sort$" has been added as a string array holding the values to be sorted. The string values are to be read from a random access text file instead of being generated by the program. In addition, line 125 sets up the ON EOF statement which will detect the end of data, set the appropriate values, and start the sort. Note also that the TIME$ function is being used. If you don't have a clock chip, you can time this yourself. One of the advantages of using a file for input is that you can make multiple runs under the same conditions to test the efficiency of particular sorting algorithms. Line 135 uses "sarray" as a pointer array to the actual position of the "sort$" values. The subroutine at line 1000 was consequently modified to use "sarray" as a pointer array, so that the strings would not have to be directly swapped:

```
1000   REM Initialize begin and end points
1005   l=1:stack(1)=n+1:m=1
1010   j=stack(l):i=m-1
```

```
 1020   IF j-m<3 THEN 1100
 1030   mid=INT((i+j)/2)
 1040   i=i+1:IF i=j THEN 1060:ELSE:IF sort$(sarray(i))<=sort$
(sarray(mid)) THEN 1040
 1050   j=j-1:IF i<>j THEN IF sort$(sarray(j))<sort$(sarray(mid)) THEN
SWAP sarray(i),sarray(j):GOTO 1040:ELSE:GOTO 1050
 1060   IF i>=mid THEN i=i-1
 1070   IF j<>mid THEN SWAP sarray(i),sarray(mid)
 1080   l=l+1:stack(l)=i:GOTO 1010
 1090   REM  check for cases of 1 or 2 elements
 1100   IF j-m<2 THEN 1130
 1110   IF sort$(sarray(m))>=sort$(sarray(m+1)) THEN SWAP
sarray(m),sarray(m+1)
 1120   REM  Set begin and end points and check for completion
 1130   m=stack(l)+1:l=l-1:IF l>0 THEN 1010
 1140   RETURN
```

Note the changes is 1040, 1050 and 1110 to have the IF statements test the proper element of "sort$" and swap the pointers if necessary. After the sort is complete, the pointer values in "sarray" are used in line 170 to look up the records in sorted order.

To effectively use this program as it stands, you need a program which will generate text files to be sorted. The program below will create a file of random "junk" which is useful to test the sort routines, and will come in handy later on in this article. It looks like this:

```
 5   OPEN#1,"JUNKFILE",12
 6   INPUT"NUMBER OF RECORDS TO CREATE: ";N
 10   FOR I=1 TO N
 15     A$=""
 20     FOR J=1 TO 5
 30       A$=A$+CHR$(65+INT(6*RND(1)))
 35       NEXT J
 38     A$=A$+" "
 41     FOR K=1 TO 4
 42       A$=A$+CHR$(48+INT(10*RND(1)))
 43       NEXT K
 45     PRINT A$
 47     PRINT#1,I;A$
 50     NEXT I
 60   CLOSE
 70   END
```

This routine is designed to create random strings with the following properties: 10 characters long, the first five characters consisting of random occurrences of

the letters "A" through "F", then a space character followed by a random four digit number. Examples of the records produced look like this:

```
DEACF 2319
ABDDC 4982
FFBBA 1965
```

A slightly fancier version would prompt for the name of the file to be created, the record lenght, etc., but this will serve nicely for the examples to follow. Generating 200 records should be sufficient for testing the sort routines. Any more would take too long without proving anything, and less would make it difficult to measure the consequences of changes to the program. For example, running 200 records through the Quicksort routine above should finish in about 40 seconds.

NOTE: Don't forget about turning the screen off during sorting! This can be done by a CTRL-5 from the keyboard, or by programatically printing an ASCII 14 to the console. The Apple III will still write information to the screen, but having it turned off speeds up operations by as much as 30 percent. This can make quite a difference when sorting or calculating, especially when you usually don't need to see the results until the operation is complete. This is also a favorite trick of people using Visicalc during "recalculates" or "loads". The console driver automatically turns the screen back on when the next input request occurs.

Living in a Tree

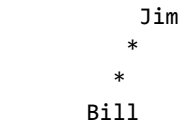
Although the Shell and Quicksort algorithms are quite efficient, they share one disadvantage which makes them difficult to use in some circumstances. There are occasions in processing data when "multi-level" sorts are desired. That is, somebody wants an address list arranged alphabetically by zip code. This means that the printout will group all people with the same zip code together, and list them alphabetically within each zip code group. To accomplish this multi-level sort, you must first sort alphabetically, and then use that order to sort everybody by zip code. This implies that each sort must perserve the physical order of the previous sort. Unfortunately, the very fact which speeds up Shell sorts and quick sorts, the swapping of data over long distances, destroys the original order of the data, making multi-level sorts impossible to implement. Our old friend, the bubble sort, does preserve order, but is impossibly slow. There are several sort techniques which solve this problem, but the one chosen for this article is the "binary tree" sort. This sort has the added virtue of very rapid insertion of additional records, once the existing records are sorted, making it also very suitable as an access method.

Before getting into the routine itself, an examination of the principles behind a binary tree data structure is a worthwhile exercise.

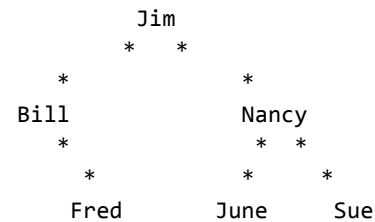
First, let's consider a list of names which we'd like to arrange in order:

```
Jim
Bill
Nancy
Fred
Sue
June
George
William
Martha
Frank
```

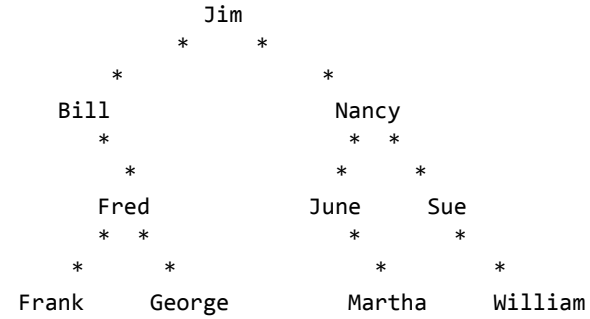
To arrange these names into a binary tree structure, we take the first name on the list as our starting point, or root. For the sake of simplicity, we'll consider this to be an upside-down tree, with the root at the top. To build branches off the tree, we take each element of the list, one at a time, and decide whether it should be connected on the right or left side of the root. For example:



Since the name "Bill" evaluates as "less than" the name "Jim", it is placed on a branch to the left. The next name, "Nancy" goes on a right-hand branch, since it is greater than "Jim". Following this logic, lets add "Fred", "Sue" and "June". The tree now looks like this:



Reviewing the process quickly, it went something like this: "Fred" was less than "Jim", but there is already an entry on the right branch, so that entry ("Bill") was examined. Since "Fred" is greater than "Bill", "Fred" was attached to a right-hand branch below "Bill". In the same way, "Sue" is greater than "Jim", so "Nancy" is checked, and since "Sue" is still greater than "Nancy", "Sue" is placed on a right-hand branch. "June", however, while being greater than "Jim", is less than "Nancy", and therefore goes to the left-hand branch. Constructing the rest of the tree with "George", "William", "Martha" and "Frank" gives a final result like this:



While all this seems lovely, and can certainly be accomplished with very few comparisons, what does it have to do with sorting? Good question! In a sense, it shares something of the technique used in quicksort, since we have partitioned the data into very small groups which have a relationship to each other. It is these interrelationships which permit building a sorted list from this structure very quickly. Two things should be obvious. No matter what value is used as a starting point, clearly the value on the extreme left-hand side of the tree is the smallest, and the value on the extreme right-hand side is the largest. To assemble the list in ascending order, then, you must first go to the left until there are no more branches leading left. In this tree, that value is "Bill". That becomes the first item in the sorted list. Next, we go down "Bill"'s right branch, to its left-most value. That's "Frank", the next item in the list. "Fred" is next, because it must be greater than "Frank" (since "Frank" was to the left of "Fred") and "George" is next, since it is the last item left on the left-hand side of the tree. That takes up to the top of the tree, where "Jim" gets added to the list, and the right-hand side is explored for its smallest (left-most) value. Following the branches, "June" is added as the next list item, and then "Martha". Since that branch is exhausted, "Nancy" is next, and then exploring "Nancy"'s other branch finds no left hand branches to add. That finishes the list, with the addition of "Sue" and "William". All that now allows us to say that the sorted order is:

```
Bill
Frank
Fred
George
Jim
June
Martha
Nancy
Sue
William
```

Although this seems like an awful lot of trouble to go to in order to sort a list of ten names, you should notice a few things which make this technique powerful. First, we are ordering the data as it is initially examined, and once we find a

place for an item, it is never moved again. Further, if the tree is relatively well balanced (more on that later), it doesn't take many comparisons to establish a place for the item. Once the "tree" is build, a sorted list can be obtained easily, and without re-examining the item values, since the position in the tree structure itself is enough to establish the order. Furthermore, if we store pointers in the tree, instead of the actual items themselves, it is not necessary to move the items at all. To show how this would work, consider the original list of names, with their associated pointers:

```
  1 = Jim
  2 = Bill
  3 = Nancy
  4 = Fred
  5 = Sue
  6 = June
  7 = George
  8 = William
  9 = Martha
 10 = Frank
```

We can assemble another list next to this one, which contains all the binary tree information, simply by indicating for each item what items are immediately below it in the tree. Using zero to indicate that the particular branch is empty, the new list would look like this:

```
item number  item value   left pointer   right pointer

    1          Jim             2              3
    2          Bill            0              4
    3          Nancy           6              5
    4          Fred           10              7
    5          Sue             0              8
    6          June            0              9
    7          George          0              0
    8          William         0              0
    9          Martha          0              0
   10          Frank           0              0
```

Graduation from B-tree University

The long-winded explanation above was designed to make you so ready to examine this month's program that your fingers itched to type it in. Wait no longer:

```
  5   REM File sort based on Binary Tree algorithm
 10   DIM parray%(1000),sortpointl%(1000),sortpointr%(1000)
 20   DIM slist%(1000),sort$(1000),stack%(200)
```

```
 25   z=0:o1=1:o2=2
```

These lines do the initialization. Note especially the "sortpointl%" and "sortpointr%" arrays. These will hold the left and right pointers described above. "Parray%" will hold the sorted list of pointers, and "slist%" holds the initial list of pointers as read from the file. "Sort$", as you might expect, holds the sort keys to be examined, and "stack%" holds temporary pointers used in assembling the b-tree structure into a sorted item list. Next comes the user input and setup section:

```
 30   HOME:PRINT"Prepare a sorted list"
 40   PRINT:INPUT"Name of file to sort: ";a$
 45   IF a$="" THEN 400
 50   IF LEN(a$)>11 THEN PRINT"Filenames must have a maximum of 11
characters":GOTO 40
 55   OPEN#1 AS INPUT,a$
 60   PRINT:INPUT"Choose the beginning and ending columns to sort on:
";b,e
 65   IF b<1 OR e<b THEN PRINT:PRINT"Invalid choice, try
again";bell$:GOTO 60
 70   ln=e-b+1
 75   OPEN#2,a$+".key"
 80   READ#2,0:IF TYP(2)<>2 THEN 90
 85   INPUT"Do you wish to sort using the existing sorted order? ";a$
 90   a$=MID$(a$,1,1)
 95   IF a$<>"y" AND a$<>"Y" THEN slist%(0)=1000:FOR i=1 TO 1000:slist%
(i)=i:NEXT:ELSE:READ#2;slist%(0):FOR i=1 TO slist%(0):READ#o2;slist%
(i):NEXT
```

Notice that the section above has a couple of interesting features. The limitation of 11 characters in the filename allows the creation of a "key" file which stores the current sorted list. This, together with the feature which allows sorting on a subset of the whole record, permits multi-level sorting to be done. Line 75 opens this file, and line 80 checks to see if there is valid data there. Line 95 then initializes the pointer array "slist%" depending on whether the sequence to be used is serial from the main file, or from the previously sorted list.

Next, let's look at the rest of the main routine:

```
 130   PRINT TIME$
 150   GOSUB 500:REM build the B-tree
 270   PRINT:PRINT
 275   PRINT TIME$
 300   GOSUB 800:REM create the sorted list
 340   PRINT:PRINT"Storing sorted list"
 350   WRITE#2,0;parray%(0)
 360   FOR i=1 TO parray%(0):WRITE#o2;parray%(i):NEXT
 370   PRINT"Sorted list stored."
```

```
 375   INPUT"Print sorted records? ";a$:IF a$<>"y" AND a$<>"Y" THEN 40
 380   FOR i=1 TO slist%(0):INPUT#o1,parray%(i);a$:PRINT a$:NEXT:GOTO 40
 400   PRINT:PRINT"End of sort program."
 410   CLOSE:END
```

As you can see, the main operations of the program are handled in the subroutine at line 500 which reads records and constructs the binary tree structure, and the subroutine at line 800 which builds the sorted pointers in the "parray%" array by decoding the structure in the b-tree. After that, the list of sorted pointers is stored in the key file, and the user is optionally allowed to list out the records. Note that although the user may have only chosen to sort on a small portion of the record, the routine at line 380 reads and prints the entire record. Next, let's examine the Binary tree build routine:

```
 490   REM Routine to build the B-tree
 500   ON EOF#1 LET slist%(0)=rec-1:POP:RETURN
 510   rec=1:GOSUB 700
 520   FOR rec=2 TO slist%(0)
 530     GOSUB 700
 540     IF sort$(rec)>=sort$(testrec) THEN 570
 550     IF sortpointl%(testrec) THEN testrec=sortpointl%(testrec):GOTO
540
 560     sortpointl%(testrec)=rec:NEXT:RETURN
 570   IF sortpointr%(testrec) THEN testrec=sortpointr%(testrec):GOTO
540
 580   sortpointr%(testrec)=rec:NEXT:RETURN

 690   REM Read a record and initialize the search
 700   INPUT#o1,slist%(rec);a$:sort$(rec)=MID$(a$,b,ln):PRINT".";
 710   sortpointl%(rec)=z:sortpointr%(rec)=z:testrec=o1
 720   RETURN
```

Notice that a subroutine at line 700 is actually used to read the data and construct the sort key. This is done to simplify changing the program to fit other data or file structures. Line 710 sets the current locations in the pointer arrays to zero and sets the initial test record to one, since testing always begins at the top of the tree. Note the use of real variables "z" and "o1" here. Not only is it faster to use variables than integer constants, but since Basic's expression evaluator works with real numbers and converts to integers only when doing the assignment to the integer variable, it is faster to use real number variables for the assignment. Using "z%" for example, would require Basic to convert "z%" to a real ("Float" it) and then convert that quantity back to an integer("Fix" it) for assignment. Now you know.

After getting the value into "sort$", lines 540 through 580 scan the binary tree structure for the appropriate place for the value. Note the double use of the NEXT statement. Executing either one will take the next value in the loop, and saves executing a GOTO. Because it is impossible to know which next will be executed last, both need to be followed by RETURN statements. It is a good idea to study the action of this routine with the b-tree data example given previously, to be sure you follow what is happening.

After the b-tree structure is built, the routine at line 800 decodes it, and builds the sorted pointer array "parray%". That routine looks like this:

```
800   parray%(0)=slist%(0)
810   recpntr=0:stackpointer=0:rec=1
820   IF sortpointl%(rec) THEN stackpointer=stackpointer+o1:
      stack%(stackpointer)=rec:rec=sortpointl%(rec):GOTO 820
830   recpntr=recpntr+o1:parray%(recpntr)=slist%(rec)
840   IF sortpointr%(rec) THEN rec=sortpointr%(rec):GOTO 820
850   IF stackpointer THEN rec=stack%(stackpointer):stackpointer=
      stackpointer-o1:GOTO 830
860   RETURN
```

Notice how this routine duplicates the algorithm we studied earlier. First, it works its way down the left side of the tree, saving records on the stack as it goes. When the left-hand pointers finally run out, that record becomes the first entry in the list (line 830) and the right side of that branch is checked (line 840). It out of right branches also, the routine exits one level up, takes the top value off the stack (line 850), decrements the stackpointer and puts that record in the list (back at line 830). Then the process continues until all values are exhausted and the return is taken at line 860. This one is worth working through too!

To "B-tree" or not to "B-tree"

Well, that concludes our look this time at binary sort techniques and b-trees. Next month we'll look at how to use this technique to create an access method which has several advantages over our "hash" algorithm from a previous episode. Its worth noting that variations on the b-tree scheme are the basis for sophisticated access methods on large machines. The Apple III Record Processing Services package uses a highly modified version of this technique as the basis for its 8-key access method. I say "highly modified" because the routine above, while it is fast and efficient for most data, has a severe problem when confronted with data which is already sorted, or nearly sorted. This is because the tree works best (needs the fewest compares) when it is relatively "balanced", that is, the data is in random order and thus falls to the left and right branches relatively equally. If the data is read in in sorted order, the result is a very long linked list, since each value will be greater than the one previous to it, and the lists will consist of all right pointers. There are techniques for

"balancing" b-trees, to solve this problem, but they were left out this time for simplicity. One technique completely outside the usual approaches is to modify the routine in line 700 to do pseudo-random reads of the data, perhaps simply starting at the middle and alternating left and right until the records are all read. Anyway, try some things out and see what you think. The "junkfile" program will allow all the practice using this routine that you can stand.

Until next month, then, tell your friends that you can't leave the house because you're too busy climbing trees. That'll perplex 'em for a while!

Exploring Business Basic, Part XVI

Catching our Breath

For the last several months we have been exploring in depth the subject of data handling. Topics like menu data entry, access methods, sorting and database programs have held sway, and have hopefully included techniques, if not whole routines that you can use in creating your own applications. This is the last month for a while that we'll take up data handling topics, because the wonderful world of Apple III graphics needs lots more attention than it has been getting on these pages.

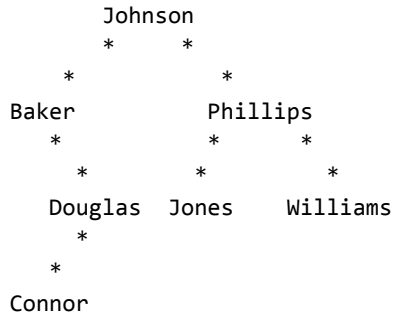
Since this is the last shot for awhile on data bases, and since last month's column promised to show how binary tree data structures could be used for an access method, this month's column includes the most ambitious program yet (at least in size). If you have been following along, you will see several old and many new techniques used in this example program. Even though it's a pretty long program, only the bare bones of a database program are there, giving you plenty of opportunities to add your own wrinkles.

Remembrance of things past

If you haven't read last month's article, you really should get familiar with it before perusing this month's missive. As a gentle reminder of last time, remember that a binary tree structure is organized in such a way that each value has associated with it a pointer to values less than the value (a "left" pointer), and a pointer to values greater than the value (a "right" pointer). the value. Because there is no way to predict the sequence in which values will be added to the tree, the individual branches may or may not contain both left and right pointers. For example, consider arranging the following list into a binary tree structure:

```
1   Johnson
2   Baker
3   Phillips
4   Jones
5   Williams
6   Douglas
7   Connor
```

The tree would look like this:



As a table of pointers, it would look like this:

		left pointer	right pointer
1	Johnson	2	3
2	Baker	0	6
3	Phillips	4	5
4	Jones	0	0
5	Williams	0	0
6	Douglas	7	0
7	Connor	0	0

There are several advantages to the Binary tree (usually called B-tree) structure. As you can see, it doesn't take many tests to find out where a value goes. This is true even if the tree is very large, as long as it is reasonably well balanced (not to many very long branches). Also, as we saw last time, it is easy to construct a sorted list from the B-tree structure, even without reference to the original values. Another advantage is the fact that we can keep as many B-trees around as memory will allow. The "Hash" method that was used some months back to make a database had the disadvantage that only one field could be a "key" field. Since we can keep multiple B-tree structures around, it is possible to have many different keys in the same file. The B-tree has some problems as an access method, however, and we'll cover them as we get into the program.

Bird's eye view of our Tree

The program below uses two files. One contains the actual data records, implemented as a single string in a random access textfile. The single string, textfile approach was chosen to keep things simple. You could easily change it by modifying the file read and write routines. Associated with the main file is a "key" file, with its name formed by appending ".key" to the end of the main file name. Information about the structure of the database is kept in the key file, along with the actual key values and b-tree pointers. At program startup, if the file you request doesn't exist, the program will allow you to define it, including

the names of the fields, which fields are to be key values, and where in the output record the field is to go. If the keyfile does exist, then all the required information is read from it, and the main file is opened for access.

One of the interesting things about the program is that it allows you to specify whether keys must be unique or not, and checks when you enter a key value to be sure. The program requires at least one key value to be unique, and uses the first such value as the key for deleting records. Examples of unique keys would be Social Security numbers, Employee numbers, etc. With some programming effort you could change this to allow choosing the record to delete from several, but the technique shown is simpler and safer. Other general capabilities including getting simple lists based on key or non-key values. The program automatically knows when you want to search on a field which is a key, and uses the fast key lookup routine. For non-key fields, the program scans the whole file looking for a match.

One last thing before we get started. This program as it stands keeps all key information in memory. This makes it fast, but limits the number of records that the program can handle. Fortunately, the Apple III has lots of memory, but even the biggest Apple III can run out if you have lots of records and keys. With a little effort, and a trade-off of size vs. performance, part of the key arrays can be kept on disk in a random access file. You should still keep as much of the first part of the key arrays in memory as possible, to reduce the number of disk accesses.

Now for the program:

```
 15   DIM item$(99),ib%(99),ie%(99),ik%(99),id%(99)
 20   DIM dup%(1000)
 25   z=0:o1=1:o2=2:bell$=CHR$(7)+CHR$(7):b20$="                    "
 28   blank$=b20$+b20$+b20$+b20$+b20$
 29   blank$=blank$+blank$+MID$(blank$,1,55)
 30   TEXT:HOME:PRINT"Database program with BSAM"
 40   PRINT:INPUT"Name of file to access: ";a$
 45   IF a$="" THEN 400
 50   IF LEN(a$)>11 THEN PRINT"Filenames must have a maximum of 11
characters":GOTO 40
 60   file$=a$
 70   GOSUB 1000
 80   IF errorcode=1 THEN 40
 85   IF errorcode=2 THEN RUN
```

The lines above do some initialization and request the filename for access.

Then a GOSUB to line 1000 does the file initialization or creation as required. Explanations of the arrays declared in lines 15 and 20 will be handled when the initialization routine is covered.

```
 90   TEXT:HOME:PRINT"Data Base: ";file$
 95   WINDOW 1,3 TO 80,24:HOME
 100   PRINT:PRINT"Functions:"
 110   PRINT:PRINT"   1 - Add a Record"
 120   PRINT"   2 - Delete a Record"
 130   PRINT"   3 - Find a Record"
 140   PRINT"   4 - List all Records"
 200   PRINT:PRINT"       Your choice: ";
 202   no.error=0
 205   INPUT"";a$:a=CONV(a$)
 210   ON a+1 GOSUB 400,2000,3000,5000,8000
 212   IF no.error THEN 90
 215   msg$="Choose a value from 1 to 4 or press RETURN to exit":GOSUB
900:PRINT CHR$(12);:GOTO 100
 400   TEXT:PRINT:PRINT"End of program."
 410   GOSUB 1500
 420   CLOSE:END
```

The lines above take care of putting up the main menu, once the file is initialized. Note that the list of functions was kept simple. It is easy to add additional routines to the menu by modifying a few lines. Note also the WINDOW statement. This method will be used extensively to keep header information on the screen during times when the display normally scrolls upward. The GOSUB in line 410 (GOSUB 1500) references the routine which saves the changes made to the file during a program run. It too will be covered in more detail later.

```
 500   FOR key=0 TO num.key-1
 510     testrec=1
 540     IF sort$(key,rec)>=sort$(key,testrec) THEN 570
 550     IF sortpl%(key,testrec) THEN testrec=ABS(sortpl%
(key,testrec)):GOTO 540
 560     sortpl%(key,testrec)=rec:GOTO 590
 570     IF sortpr%(key,testrec) THEN testrec=ABS(sortpr%
(key,testrec)):GOTO 540
 580     sortpr%(key,testrec)=rec
 590     NEXT
 595   RETURN
```

For those of you who were tuned in last time, the routine in lines 500-595 above should look familiar. Last month's version handled only one key, while this one uses a two-dimensional sort value array and pointer arrays to update multiple keys. Another change is interesting, for you sharp-eyed routine-watchers. Note that line 550 and 570 assign the Absolute Value (ABS) of the pointer array to the variable 'testrec'. This precaution was taken because, as we shall see later, a negative pointer is used as an indication that the given value has been deleted, even though the value itself must remain to complete the b-tree.

```
 600   testrec=1:dup=0:errorcode=0:del.rec=(sortpl%(key,0)<0)
 610   IF key$>=sort$(key,testrec) THEN 630
 615   del.rec=(sortpl%(key,testrec)<0)
 620   IF sortpl%(key,testrec) THEN testrec=ABS(sortpl%
(key,testrec)):GOTO 610
 625   RETURN
 630   IF key$<>sort$(key,testrec) THEN 640
 635   IF NOT del.rec THEN dup=dup+1:dup%(dup)=testrec
 640   del.rec=(sortpr%(key,testrec)<0)
 645   IF sortpr%(key,testrec) THEN testrec=ABS(sortpr%
(key,testrec)):GOTO 610
 650   RETURN
```

The routine from 600-650 is a variation on the binary tree search routine in the previous example, except that its sole function is to assemble a list of record numbers whose key values match the variable "key$". These are stored in the array "dup%". Note that the variable "del.rec" is used as a flag to ignore a record if its pointer is negative (deleted). This routine is used by the "Find" function to scan the file for matching key values and return all records which apply.

```
 700   testrec=1:errorcode=0:del.rec=(sortpl%(key,0)<0)
 710   IF key$>=sort$(key,testrec) THEN 730
 715   del.rec=(sortpl%(key,testrec)<0)
 720   IF sortpl%(key,testrec) THEN testrec=ABS(sortpl%
(key,testrec)):GOTO 710: ELSE:RETURN
 730   IF key$=sort$(key,testrec) AND NOT del.rec THEN
errorcode=1:RETURN
 735   del.rec=(sortpr%(key,testrec)<0)
 740   IF sortpr%(key,testrec) THEN testrec=ABS(sortpr%
(key,testrec)):GOTO 710: ELSE:RETURN
```

The routine above (700-740) is the most specialized of all. Its sole function in life is to check to see if a given key value has a duplicate value already in the file. It is used to insure that the keys marked "no duplicates" are in fact, unique.

```
 900   VPOS=21:HPOS=1:INVERSE:PRINT msg$;CHR$(31);:IF beep THEN PRINT
bell$;
 905   VPOS=line:HPOS=col:NORMAL:RETURN
 910   VPOS=21:HPOS=1:NORMAL:PRINT CHR$(31);:VPOS=line:HPOS=col:RETURN
 930   FOR i=1 TO delay*60:PRINT CHR$(22);:NEXT:RETURN
```

The routines in lines 900-930 above are utilities used throughout the program.

Line 900-905 puts a message in the message window and restores the cursor.

Line 910 clears the message window, and 930 creates a delay of "delay" seconds, by printing screen sync characters (1/60 of a second each). Remember that delays on the Apple III should be programmed like this, rather

than with FOR-NEXT loops. Because the Apple III is interrupt driven, it is really impossible to tell exactly how long a given routine will take to execute.

Now for the fun stuff:

```
 1000   REM initialize file
 1005   errorcode=0
 1010   OPEN#2,file$+".key"
 1020   READ#2,0:IF TYP(2)<>1 THEN 1100
 1030  READ#2;num.rec,num.key,pl.rec,pr.rec,sort.rec,num.item,
item.rec,rec.len,tot.rec
 1032   OPEN#1,file$,rec.len
 1033   DIM sortpl%(num.key-1,1000),sortpr%(num.key-1,1000), sort$
(num.key-1,1000)
 1035   IF num.rec=0 THEN 1092
 1037   IF TYP(1)<>8 THEN PRINT"Your file has been damaged.";bell$:
errorcode=2: IF TYP(1)=0 THEN DELETE file$:RETURN:ELSE:RETURN
 1040   READ#2,pl.rec
 1050   FOR i=1 TO num.rec:FOR j=0 TO num.key-1:READ#2;sortpl%
(j,i):NEXT:NEXT
 1060   READ#2,pr.rec
 1070   FOR i=1 TO num.rec:FOR j=0 TO num.key-1:READ#2;sortpr%
(j,i):NEXT:NEXT
 1080   READ#2,sort.rec
 1090   FOR i=1 TO num.rec:FOR j=0 TO num.key-1:READ#2;sort$
(j,i):NEXT:NEXT
 1092   READ#2,item.rec
 1095   FOR i=1 TO num.item:READ#2;item$(i),ib%(i),ie%(i),ik%(i),id%
(i):NEXT
 1097   RETURN
```

You guessed it, the initialization routine! The first step is to open the "key" file, formed by adding ".key" to the file name in line 1010. If it exists and contains data, then initialization proceeds. If not (line 1020), then the program jumps to line 1100, where the new file is created. Line 1030 reads a number of important variables from the key file. Most of them are self explanatory. Variables ending in ".rec" point to the beginning record numbers in the key file where the associated arrays are to be found. Thus "pl.rec" is the record number where the "sortpl%" array is to be found, "item.rec" points to where the lists of data item definitions start. The exception to this rule is "tot.rec" which is simply the total of valid (undeleted) records in the file, different from num.rec, which is the total number of physical records. After reading this in, the main file is opened in 1032 using the record length read from the key file. Line 1033 then dimensions the appropriate arrays according to the number of keys defined in the key file. Note that this is an extremely powerful capability, not found in many versions of BASIC. The arrays have been arbitrarily defined to be 1000 records long. If you

have a 128K system and use lots of keys, you may want to reduce this total number. The rest of the routine determines if it is necessary to read in the key data, and if so, does it in lines 1050-1095. We'll cover the meaning of the arrays in line 1095 below, when the creation routine is covered.

```
1100   PRINT"The file ";file$;" is not a database file"
1110   INPUT"Do you wish to make it a database file? ";a$
1120   a$=MID$(a$,1,1):IF INSTR("Yy",a$) THEN 1200
1140   DELETE file$+".key":errorcode=1:RETURN
```

The section above sets up for creation of a new file. Note the use of INST in line 1120. It substitutes for 'IF a$="Y" or a$="y" THEN 1200'. Another way to write line 1120 is :

```
1120  IF INSTR("Yy",MID$(a$,1,1)) THEN 1200
```

```
1200   HOME:PRINT"Database setup - Record definition:"
1210   PRINT:WINDOW 1,3 TO 80,24:HOME
1230   FOR i=1 TO 99
1235     IF VPOS=22 THEN PRINT:VPOS=21
1240     PRINT USING 1245;i;:line= VPOS
1245     IMAGE "Item "  ,2#," - Name: "
1250     INPUT"";item$(i):IF item$(i)="" THEN 1370
1252     HPOS=17:VPOS=line:PRINT MID$(item$(i),1,16);:PRINT CHR$(31);
1255     IF LEN(item$(i))>16 THEN PRINT bell$;:HPOS=1:GOTO 1240
1260     VPOS=line:HPOS=34
1270     INPUT"begin: ";a$
1280     ib%(i)=CONV%(a$)
1285     IF ib%(i)<1 THEN PRINT bell$;:GOTO 1260
1290     VPOS=line:HPOS=41:PRINT USING"2#";ib%(i);
1300     INPUT"  end: ";a$
1310     ie%(i)=CONV%(a$)
1315     IF ie%(i)<ib%(i) THEN PRINT bell$;:GOTO 1290
1320     VPOS=line:HPOS=50:PRINT USING"2#";ie%(i);
1330     INPUT"  Key? ";a$
1340     ik%(i)=(INSTR("Yy",MID$(a$,1,1))>0)
1350     VPOS=line:HPOS=59:PRINT MID$("NY",ik%(i)+1,1);:PRINT CHR$(31);
1351     IF NOT ik%(i) THEN id%(i)=1:PRINT:GOTO 1360
1352     INPUT"  Duplicates? ";a$
1355     id%(i)=(INSTR("Yy",MID$(a$,1,1))>0)
1357     VPOS=line:HPOS=74:PRINT MID$("NY",id%(i)+1,1);:PRINT CHR$(31)
1360     NEXT i
```

The routine above is a rather elaborate input routine which prompts for each field name and gets beginning and ending columns, whether the field is to be a key, and if it is a key, whether duplicate values are allowed. Note the extensive use of VPOS and HPOS to facilitate editing, and the use of INSTR and MID$ in

lines 1340 thorough 1357 to save time and program size. As can be seen by examination, "item$" holds the individual field names, "ib%" and "ie%" hold the beginning and ending field positions (and thus the maximum field size), and "ik%" and "id%" hold the flags for key fields and duplicates allowed. If memory size is a problem, these two arrays could be combined with a minimal amount of programming effort.

```
 1365   msg$="Initializing file '"+file$+"'.":GOSUB 900
 1370   num.item=i-1:num.key=0
 1375   FOR i=1 TO num.item:IF ie%(i)>rec.len THEN
rec.len=ie%(i):NEXT:ELSE:NEXT
 1380   FOR i=1 TO num.item:IF ik%(i) THEN
num.key=num.key+1:NEXT:ELSE:NEXT
 1385  num.rec=0:pl.rec=100:pr.rec=200:sort.rec=300:item.rec=10:
rec.len=rec.len+1
 1390  WRITE#2,0;num.rec,num.key,pl.rec,pr.rec,sort.rec,num.item,
item.rec,rec.len,num.rec
 1395   WRITE#2,300;0:REM establish end of file
 1400   READ#2,item.rec
 1410   FOR i=1 TO num.item:WRITE#2;item$(i),ib%(i),ie%(i),ik%(i),id%
(i):NEXT
 1420   msg$="File '"+file$+"' is initialized":GOSUB 900:delay=2:GOSUB
930
 1430   OPEN#1,file$,rec.len
 1435   DIM sortpl%(num.key-1,1000),sortpr%(num.key-1,1000),sort$
(num.key-1,1000)
 1440   TEXT:HOME:RETURN
```

The lines above take the information from the creation routine and write it to the key file, open the main file, and dimension the appropriate arrays for use by the program.

```
 1500
WRITE#2,0;num.rec,num.key,pl.rec,pr.rec,sort.rec,num.item,item.rec,
rec.len,tot.rec
 1510   IF num.rec=0 THEN 1600
 1520   READ#2,pl.rec
 1530   FOR i=1 TO num.rec:FOR j=0 TO num.key-1:WRITE#2;sortpl%
(j,i):NEXT:NEXT
 1540   READ#2,pr.rec
 1550   FOR i=1 TO num.rec:FOR j=0 TO num.key-1:WRITE#2;sortpr%
(j,i):NEXT:NEXT
 1560   READ#2,sort.rec
 1570   FOR i=1 TO num.rec:FOR j=0 TO num.key-1:WRITE#2;sort$
(j,i):NEXT:NEXT
 1600   PRINT"File '"file$"' updated.  There are ";tot.rec;" records in
the file."
```

```
 1610   RETURN
```

Lines 1500 through 1610 do just the opposite, storing away all the current data about the key file onto the appropriate records.

```
 2000   TEXT:HOME:PRINT"Add a Record to file '";file$"'."
 2010   PRINT
 2012   WINDOW 1,3 TO 80,24:HOME
 2015   rec=num.rec+1:key=-1:line$=MID$(blank$,1,rec.len-1)
 2020   FOR i=1 TO num.item
 2022     beep=1:IF ik%(i) THEN key=key+1
 2025     field.len=ie%(i)-ib%(i)+1
 2035     IF VPOS>20 THEN PRINT:PRINT:VPOS=20
 2040     PRINT"("i") ";item$(i)": ";
 2045     line= VPOS:col= HPOS
 2050     INPUT"";a$
 2060     IF a$="" AND i=1 THEN 2200
 2070     IF LEN(a$)>field.len THEN msg$="Entry is too long":GOSUB
900:GOTO 2050
 2075     GOSUB 910
 2080     IF NOT(ik%(i) AND NOT id%(i)) THEN 2100
 2085     key$=a$:GOSUB 700
 2090     IF errorcode THEN msg$="Entry must be a unique value in this
field":GOSUB 900:GOTO 2050
 2095     GOSUB 910
 2100     SUB$(line$,ib%(i),field.len)=a$
 2110     IF ik%(i) THEN sort$(key,rec)=a$
 2115     PRINT
 2120     NEXT i
 2130   msg$="Record being added.":beep=0:GOSUB 900
 2140   PRINT#1,rec;line$
 2150   IF rec>1 THEN GOSUB 500
 2155   num.rec=rec
 2157   tot.rec=tot.rec+1
 2160   GOSUB 910
 2165   PRINT:PRINT
 2170   GOTO 2015
 2200   IF VPOS>20 THEN PRINT:PRINT:VPOS=20
 2202   PRINT"End of Add.  ";tot.rec" records now in file '"file$"'."
 2205   msg$="Press return to continue: ":GOSUB 900:GET a$
 2210   no.error=1:TEXT:RETURN
```

The Add routine above is long, but relatively straightforward. Notice that lines 2080-2095 check for unique values it required by using the routine at line 700. Notice also how the record is built up by using SUB$ to insert fields into the

"line$" string. After all fields are entered, a GOSUB to line 500 is performed to add all the keys to the pointer arrays.

```
 3000   TEXT:HOME:PRINT"Delete a Record in file '"file$"'."
 3010   PRINT
 3020   WINDOW 1,3 TO 80,24
 3025   unique=0
 3030   FOR i=1 TO num.item:IF NOT ik%(i) OR(ik%(i) AND id%(i)) THEN
NEXT:ELSE:unique=i
 3035   HOME
 3040   PRINT:PRINT"Records are deleted by using the '"item$(unique)"'
field"
 3050   PRINT:PRINT item$(unique)": ";
 3060   INPUT"";a$
 3070   field.num=unique:key=-1:FOR i=1 TO field.num:IF ik%(i) THEN
key=key+1:NEXT:ELSE:NEXT
 3080   IF a$="" THEN 3400

 3090   HOME
 3100   msg$="Searching for "+item$(field.num)+": "+a$:line= VPOS:col=
HPOS:GOSUB 900
 3105   key$=a$
 3110   GOSUB 600
 3115   IF NOT dup THEN 3550
 3120   rec=dup%(1)
 3125   GOSUB 5620
 3130   IF VPOS>17 THEN PRINT:PRINT:PRINT:PRINT:PRINT:VPOS=17
 3135   PRINT:PRINT"The record is:"
 3140   GOSUB 5650
 3142   IF VPOS>19 THEN PRINT:PRINT:PRINT:VPOS=19
 3145   PRINT:PRINT"Delete? ";
 3150   msg$="Type 'Y' to Delete, any other key to Retain:":line=
VPOS:col=HPOS:GOSUB 900
 3155   INPUT"";a$
 3160   IF NOT INSTR("Yy",MID$(a$,1,1)) THEN 3300
 3165   msg$="Deleting the Record":line= VPOS:col= HPOS:GOSUB 900

 3170   IF rec=1 THEN FOR i=0 TO num.key-1:sortpl%(i,0)=-1: sortpr%
(i,0)=-1:NEXT :GOTO 3200
 3175   FOR i=0 TO num.key-1:FOR j=1 TO num.rec
 3180       IF sortpl%(i,j)=rec THEN sortpl%(i,j)=-rec:GOTO 3195
 3185       IF sortpr%(i,j)=rec THEN sortpr%(i,j)=-rec:GOTO 3195
 3190       NEXT j
 3195     NEXT i
 3200   GOSUB 3500
```

```
 3205   msg$="Record Deleted":line= VPOS:col= HPOS:GOSUB 900
 3210   delay=2:GOSUB 930
 3215   tot.rec=tot.rec-1
 3220   GOTO 3000

 3300   msg$="Record not Deleted.":line= VPOS:col= HPOS:GOSUB 900
 3310   delay=2:GOSUB 930
 3320   GOTO 3000

 3400   no.error=1:RETURN

 3500   PRINT#1,rec;" "
 3510   RETURN

 3550   msg$="Record not found":line= VPOS:col= HPOS:GOSUB 900
 3560   delay=2:GOSUB 930
 3570   GOTO 3000
```

"Delete" above, is much tougher technically. Because the B-tree depends on an ordered structure of key values, its not possible to simply blank out a value in "sort$" and zero out the pointers in the arrays. The full solution is more complex than is worth delving into here (read about "balanced b-trees" and "B-splat" trees in references). To keep things simple, we just negate the pointers and go on. This is done in 3170-3220.

```
 5000   TEXT:HOME:PRINT"Find Records in file '";file$;"'"
 5010   PRINT
 5020   WINDOW 1,3 TO 80,24
 5025   unique=0
 5030   FOR i=1 TO num.item:IF NOT ik%(i) OR(ik%(i) AND id%(i)) THEN
NEXT:ELSE:unique=i
 5035   HOME
 5040   PRINT:PRINT"Functions:"
 5050   PRINT:PRINT"   1 - Search on a single field value"
 5070   IF unique THEN PRINT"   2 - Find a record using the '"item$
(unique)"'field"
 5080   PRINT:PRINT"       Your Selection:";
 5090   INPUT"";a$
 5110   a=CONV(a$)
 5120   ON a+1 GOTO 5900,5200,5500
 5130   msg="Choose a number from 1 to 2 or press RETURN to exit":GOSUB
900:PRINT CHR$(12);:GOTO 5040
```

The lines above are the start of the rather long "Find" routine, which gives the option of searching on an individual field or using the first unique field, the same one used by delete. This second option was put in for convenience, since the

same thing can be accomplished with option 1 and a little more effort. An interesting option that could be added would be to search on combinations of fields.

```
 5200   HOME
 5205   PRINT:PRINT"Search on a single field value":PRINT
 5210   FOR i=1 TO num.item
 5215     IF VPOS>20 THEN PRINT:PRINT:VPOS=20
 5220     PRINT USING 5225;i,item$(i)
 5225     IMAGE "(",2#,")",2x,16a
 5230     NEXT i
 5240   PRINT:PRINT"       Your Selection: ";
 5250   INPUT"";a$
 5260   a=CONV(a$)
 5270   IF a=0 THEN HOME:GOTO 5040
 5280   IF a<1 OR a>num.item THEN msg$="Field number invalid":GOSUB
900:PRINT
CHR$(12);:GOTO 5205
 5282   select.all=0
 5283   field.len=ie%(a)-ib%(a)+1
 5284   field.num=a
 5285   PRINT:PRINT"Field value: ";
 5286   msg$="Use '=' for all, '>' for all non-blank":line= VPOS:col=
HPOS:GOSUB 900
 5287   INPUT"";a$
 5288   IF a$="" THEN PRINT CHR$(12);:GOTO 5205
 5289   IF MID$(a$,1,1)="=" THEN select.all=1:ELSE:IF MID$(a$,1,1)=">"
THEN select.all=2
 5290   IF LEN(a$)<field.len THEN value$=a$+MID$(blank$,1, field.len-
LEN(a$)):ELSE:value$=MID$(a$,1,field.len)
 5291   rvalue$=a$

 5292   HOME
 5295   IF ik%(field.num) AND NOT select.all THEN 5400
 5300   msg$="Scanning the file":line= VPOS:col= HPOS:GOSUB 900
 5305   rec.found=0
 5307   IF tot.rec=0 THEN 5360
 5310   FOR rec=1 TO num.rec
 5320     GOSUB 5600
 5325     IF no.rec THEN 5350
 5330     GOSUB 5475
 5335     IF VPOS>19 THEN PRINT:PRINT:PRINT:VPOS=19
 5340     IF select THEN GOSUB 5650
 5350     NEXT rec
 5355   IF rec.found THEN msg$="No more records, Press RETURN to
```

```
continue:":GOSUB 900:GET a$:GOTO 5035
 5360   msg$="No records found":GOSUB 900:delay=2:GOSUB 930:GOTO 5035
```

Lines 5200-5360 handle the case of searching on a given field. Notice that there are additional options of selecting all records, or all records with non-blank fields. In addition, line 5295 checks to see if the field is a key field, and if so, jumps to the routine below which does a fast scan of the key in memory. Notice also that all actual I/O is done through subroutines in lines 5600 and 5650, to facilitate changing file structures with a minimum of effort.

```
 5400   key=-1:FOR i=1 TO field.num:IF ik%(i) THEN
key=key+1:NEXT:ELSE:NEXT
 5405   key$=rvalue$
 5410   msg$="Scanning the Key file":line= VPOS:col= HPOS:GOSUB 900
 5415   GOSUB 600
 5417   IF NOT dup THEN 5360
 5420   FOR i=1 TO dup
 5425     rec=dup%(i)
 5430     GOSUB 5620
 5432     IF no.rec THEN 5445
 5435     IF VPOS>19 THEN PRINT:PRINT:PRINT:VPOS=19
 5440     GOSUB 5650
 5445     NEXT i
 5450   GOTO 5355
```

This above is the routine used to scan the key file for a value. Notice that it uses the subroutine at line 600 to pull duplicates of a given value. Then the "dup%" array is used as the record list.

```
 5475   select=0
 5480   IF select.all=1 THEN select=1:RETURN
 5485   IF select.all=2 AND field$>="!" THEN select=1:RETURN
 5490   IF field$=value$ THEN select=1:RETURN
 5495   RETURN
```

This is a routine which is used by the search routine to determine if the field meets the search criteria.

```
 5500   field.num=unique:key=-1:FOR i=1 TO field.num:IF ik%(i) THEN
key=key+1:NEXT:ELSE:NEXT
 5505   PRINT:PRINT"       "item$(field.num)": ";
 5510   INPUT"";a$
 5515   IF a$="" THEN PRINT CHR$(12);:GOTO 5205
 5520   HOME
 5525   msg$="Searching for "+item$(field.num)+": "+a$:line= VPOS:col=
HPOS:GOSUB 900
 5530   key$=a$
 5535   GOTO 5415
```

Line 5500 sets up the search for the first unique field, and then uses the regular keysearch routine to complete.

```
 5600   INPUT#1,rec;line$
 5601   no.rec=0
 5602   IF LEN(line$)<rec.len-1 THEN no.rec=1:RETURN
 5605   field$=MID$(line$,ib%(a),field.len)
 5610   RETURN

 5620   INPUT#1,rec;line$
 5622   no.rec=0
 5625   IF LEN(line$)<rec.len-1 THEN no.rec=1:RETURN
 5630   RETURN

 5650   PRINT"("line$")"
 5655   rec.found=1:RETURN

 5900   no.error=1:RETURN
```

The routines above are general purpose and used by parts of the search routines and others to perform actual read operations on the files.

Which brings us at long last to the last routine (at least for this article!).

```
 8000   TEXT:HOME:PRINT"List all records in file '";file$"'."
 8005   WINDOW 1,3 TO 80,24:HOME
 8007   IF tot.rec=0 THEN 8035
 8010   FOR rec=1 TO num.rec
 8015     GOSUB 5620
 8017     IF no.rec THEN 8030
 8020     IF VPOS>19 THEN PRINT:PRINT:PRINT:VPOS=19
 8025     GOSUB 5650
 8030     NEXT rec
 8035   msg$=CONV$(tot.rec)+" records listed. Press RETURN to
continue:":GOSUB 900:GET a$
 8040   no.error=1:RETURN
```

Lines 8000-8040 provide a quick list of all records using the previously defined read routines.

There! More that anyone wants to know about B-tree access methods in BASIC. Purists among you will note that alot has been left to the imagination. For example, what happens when the key array fills up with deleted records? How fast will this method add records when there are lots of records and lots of duplicates? These are real questions, and are solveable, with effort and cleverness. It was not the intention of this article to give you a working general purpose database program. There are plenty of those on the market for the Apple III. Rather, the program has given us a chance to explore programming

techniques which may prove very useful in specific tasks, and should enrich your knowledge of programming in general. It is therefore with misty eyes that we bid databases a fond, albeit temporary, farewell.

Greener Pastures

Next time we will start a new series on Apple III graphics capability, beginning with something that most people think is too hard: a high-speed, high-res game for the Apple III in BASIC! Until then, keep pounding on your Apple III.

Exploring Business Basic, Part XVII

An Immediate Apology

If, like a lot of people, you've looked at the program listing in this article before reading the text, you're probably wondering "Where is that Hi-res game he promised last time? This giant mess can't be it!". Right again, buckaroo. The game that was promised will be delivered, but next month. A bigger issue, related to hi-res games, will be covered this month as a precursor (heh,heh) to that article. To tell the truth, the original plan was to create a little Shape generator and editor to do the graphics animation characters. Well, the shape editor grew and grew, and threatened to overwhelm the entire article. Shortly after threatening to do so, it did, and forthwith, this month's article presents a hi-res character set, shape and font editor with some really nice features. Next month we'll use the editor to create creatures to inhabit our game. Also, because the program is so large, the usual chatty narrative will be somewhat terse. And now, on with it!

An Immediate Digression

Having made all those imposing statements above about terseness (tersity?), Several new products have been introduced on the Apple III which deserve notice. First is a parallel printer interface from Interactive Structures. What distinguishes this card is the software, which is really complete. It supports many different printers, including Apple's new Dot Matrix Printer, both in emulation mode and with SOS drivers. The driver can print with a variety of options, and if your printer has a graphics print option, the driver can even use the current screen font for printing! In addition, an invokable module is supplied, usable from Basic and Pascal, which permits graphics screen dumps with lots of options. Altogether, a nice piece of work.

The next two products are floppy disk drives for the Apple III. Yes, Virginia, there are high-density floppies! Apple introduced their new UniFile and DuoFile at Comdex, and they should be available soon. They feature 860K per diskette at a very reasonable price. Also on the market now is the MicroSci A143, a 560K disk which daisy-chains along with existing Disk III's. This disk has less storage, but has the advantage of not requiring a slot. Since both disks come with SOS drivers, they are completely compatible with all your other software. Go SOS!

And Now, On With the Show

First, we'll look at the general operation of the shape editor, using line number ranges to describe large operations. Then, some of the routines will be examined in detail to clarify points of possible confusion and to indicate which routines could be adapted for other purposes.

General Operation

Apple III high-resolution shapes and characters are drawn on the screen using a procedure in the BGRAF invokable module called DRAWIMAGE. This actually utilizes the DRAWBLOCK capability of the .GRAFIX driver. Unfortunately, knowing all this, and even reading all the documentation, doesn't make it completely clear. The program below should help by illustrating lots of useful subroutines which perform these functions.

The program operates by creating a work area on the screen that allows you to look at and change data blocks used by DRAWIMAGE. These data blocks consist of integer arrays which DRAWBLOCK interprets as bits to be drawn on the screen. From now on, the word "Bit" will mean a piece of data in an array, and the word "Pixel" will mean the representation of that bit as a dot on the screen. Obviously, the different graphics modes have different looking pixels, although the bit in the array is the same. For now, we won't worry about color, since that is not a function of the bit arrays, but rather the pencolor assigned at the time the bits are drawn.

Another important feature of the editor is that is maintains separate windows on the screen for each video mode. This allows you to see the effect of the shape as it's being created. Sometimes a shape that looks good in one mode looks terrible in another, because of the different proportions. Enough theory, let's look at the code!

Getting a Bit Under Control

After setting up arrays in lines 10-15, variable and table initialization is done in lines 4000-4500. The program uses several arrays as work areas and holding areas for data, and others for fast lookup of information for performance. "Work%" is an array which holds the bit patterns currently available to be modified. These can come from a character set ("char%"), a shape definition ("shape%") or a Apple III system font ("cset%"). The most important tables used for lookup are "shex%", "bits%" and "flip".

"Shex%" is defined in lines 4000 through 4015 and contains the bit representations of all 16 HEX digits in four modes: two high, four wide; one high, four wide; one high, two wide and one high, one wide. These modes

correspond to character bit patterns for the Work Area, 140X192 mode, 280X192 mode and 560X192 mode respectively. This is necessary since we will be using the 560X192 screen for all editing functions, but will want to look at the characters and shapes as they would appear in the other modes.

"Bits%" is a table which has four entrys for each HEX digit (one for each binary bit) and allows quick determination if a particular bit is on or off in a given hex number.

"Flip" contains 256 entrys, each one corresponding to a byte with its bits reversed end for end. For example, consider the number 75. In HEX, it would be "4B", in binary "01001011". If we were to flip the bits exactly, the result would be "11010010", HEX "D2", or decimal 210. All this would be extremely unimportant if it were not for the fact that the character images used by the Apple III system fonts and the images used by the BGRAF invokable module are exactly reversed. Therefore, to move back and forth between the two requires some way of reversing the sets. Thus the table "flip". By looking up the entry under 75, the program will find the value 210 and, making the substitution, will flip the byte. Line 4060 builds this array from a smaller array called "lookup" which consists of "flipped" HEX digits.

Once initialization is done, lines 30 through 50 do some further setup, and the program proceeds to build the graphics screen for editing. If you are wondering whether the whole program is worth entering, try typing in lines 5-200, just to get a look at the screen. It'll make a lot more sense out of the discussion to follow. Notice that line 100 refers the the subroutine at line 600, which creates the four windows referred to earlier. This routine is also used later to clear the windows quickly.

Once the screen is initialized, lines 210-235 get the command and dispatch to the proper routine for processing. Note that the actual input is handled in a subroutine at line 3000. This routine, along with the error routine at line 3070 and the message routine at line 3100, handle character input and output to the graphics screen. Remember, the primary action is on the graphics screen so we want to avoid flipping back and forth between graphic and text screens. You could use this routine in any program that wants to accept text input on the graphics screen.

A Routine a Day

Rather than describe the various functions one at a time, it would be more instructive to look at some in detail and give a general overview of the rest. One command that shows off most of the features of the program is Load, selected as item 2 on the menu. Load is handled by the routine at 1400.

Getting Loaded in Hi-res

First the routine prompts for what kind of file to load. Shape and Character set files are unique to this program, but the Font file must be treated specially, since Basic cannot directly open a System Font file. Note the use of the INSTR function at line 1410 to determine the value of "choice". There are two spaces in front of the "Ss" and one space between "Ss", "Cc" and "Ff". When divided by 3 and truncated (INT), the result is 0, 1, 2, or 3. This is a handy technique to handle multiple choice options in either upper or lower case.

ON ERR is set in line 1430 to handle any errors in dealing with the files, and then, unless the file is a font file, it is opened in line 1435. If the choice is a font file, the "Getfont" invokable proceedure is used to load it into memory, and the subroutine at line 3950 is called to "flip" the font to the graphics mode. If the choice is a shape or character file, then information about the data is read in line 1450 from the first record. "Filtyp" is the type code used to save the file, "ch" is character height, "cw" is character width, and "sl" is the valid length of the shape definition in words (0-7). Not all these values will have meaning, depending on the value of "filtyp". If everything is ok, then the "Filread" invokable proceedure is called (from the "REQUEST.INV" module) to read in the array from the file. The actual size of the file as read ("ret%") is checked against the expected size ("size%(filtyp)") and if everything checks out, then the subroutine at line 3600 is called to display the results of the load.

The Subroutine at line 3600 does most of the work of displaying the bit images on the screen as various size pixels. First, depending on the type of image to be displayed (shape, character set or font) it loads a section of the appropriate array into the work area using the routines at lines 3700-3940. If the choice is a shape, it is directly transferred to the work area, since shape definitions are arbitrarily defined to be a maximum of 128 bits wide by 16 bits high. In the case of character set and font definitions, the routine at line 3800 prompts for a starting character number to display in the work area. Normally character definitions are each 8 bits wide, and fonts are always 8 bits wide. Although it is possible to define a larger character cell size, for the purposes of this program fonts will be transferred to the work area on 8 bit boundarys, and character sets will be transferred on even 8 bit boundarys. This simplifies things considerably, since the data is stored in integer (16 bit) arrays. Lines 3815-3830 determine the starting location in the "char%" array to begin the transfer, and calculate "sl", the "shape length" which is the number of array elements (maximum 8 elements or 128 bits) to display in the work area. As you can see, the storage format of shapes and character set definitions is similar. Basically, the first index of the array represents the bits in a given row, and the second index represents the row number. Things are considerably different in the font definition, however, as

shown in the routine at line 3900. The "Getfont" proceedure reads the font definition in into a one-dimensional array which is decoded in lines 3915 to 3935. You can think of the font definition as a set of 8 bytes for each character, one byte for each character row, arranged one after another. Each 8 bytes (4 integer elements), a new character begins. The requirements of the Drawimage proceedure are that each row byte be in a separate row element, and that each row element (an integer) contain the two row bytes of two adjacent characters. Whew! No wonder a lot of these programs haven't been written! Anyway, trust it, it works.

See it all

After all that messing around, we now have the proper information in the "work%" array, and can draw the images on the screen with the routines at lines 3605-3690. After clearing all the windows, and setting up the variables ("rs" - starting row to display, "re" - ending row to display, "bw" - beginning word of column, "ew" - ending word of column), we are ready to draw in each window. Line 3610 gives the starting position of the window in "xdot" and "ydot" and then defines where in the "shex%" array that the drawing of the pixel definitions will take place. Rows zero and one of "shex%" are pixel definitions of hex digits in 4 wide, 2 high format, the format for the work area display. The subroutine at line 3670 then proceeds through the work% array, drawing from the definitions in "shex%". Note that line 3625 performs the Drawimage proceedure directly, since the window at 7,117 is for 560X192 mode, which is the current screen mode, and can thus be drawn directly. The drawing proceeds with list 3630-3635, which sets up the 280X192 mode (2 wide, 1 high) and finally line 3640-3645, the 140X192 mode (4 wide, 1 high) and the display is complete.

Although the information above is useful, it is by no means complete. A through reading of Appendix I of the Basic manual on the BGRAF invokable, and reading of the Standard Device Drivers manual section on .GRAFIX is strongly recommended.

This program has also been somewhat simplified by limiting the shape and character size definitions in size. In actual practice, Drawimage can be used to draw shapes or characters actually larger than the entire graphics screen! An interesting challenge for you is to modify this routine to handle larger shape and character definitions, treating the work area as a window, as is done to a limited extent with the character set and font definitions.

Putting the Bits and Bytes to Bed

A quick look at the Save function is worthwhile, especially now that you are familiar with the internal format of the information. The save routine is found at lines 1100-1195 and is relatively straight-forward except for lines 1170-1180. In

line 1170 a check is made for file type 3, the font file. Font files are saved without accompanying information, since the format is fixed. This also allows you to alter the file type on disk to type "FONT" and load the font into the standard system character set. The Pascal filer will allow this, and there is a new invokable from Foxware (reviewed next month) which makes it easy to do from Basic. In addition, the "Loadfont" proceedure will allow you to use "Download.inv" for the same purpose.

Line 1175 then goes to a subroutine, depending on file type, which loads the work area back into the appropriate array for saving. Any modifications, as we will see later, are made only in the work area, until saved, or another work area is chosen. After transfering the work area, a check is made to see if the Save is being attempted in a different format than the original Load. For type 1 (shape) the sizes are identical to the work area, so nothing has to be done, but for character set to font and vice-versa, some translations must be made. They are handled by the subroutines at 2000, and then the appropriate array is written to disk, safe at last.

Other Interesting Stuff

Catalog, Delete and Define are relatively simple and won't be covered here. View uses some of the functions we have already discussed in Load and Save, and permits scanning around in the character set or font, beginning at different places. This routine first must save the current work area back to its original array, and then load and display the new section, much the same as the original Load did. Obviously for shapes, there is nothing to view beyond what is on the screen, so a redisplay is done.

Ok, its there, Now what do I do?

Which brings us to Edit, Clear and Invert.

After displaying what you want to edit in the work area, selecting option 4 gets you into the routine at line 250. An immediate GOSUB is performed to line 450 which determines if the bit at the current grafix cursor location ("chorz","cvert") is on or off. This value is stored in "cstate". On returning "Cflash" is set to the opposite value, and and ON KBD loop is entered to flash a pixel at that location. Note that the routine in line 275 makes a longer wait between flashes if cstate is 0, allowing you to tell whether the underlying pixel is on or off. When a key is pressed on the keyboard, the long routine at line 280-345 is entered to process the keystroke and perform the appropriate action. The request is decoded in line 295 by scanning the "ctrl$" string, previously defined in line 4085. Then line 300 transfers control to the appropriate routine. Lines 315-330 handle simple cursor movements. Note that by holding down the open-apple key, the value of "skp" is set to the current character width (line 290), useful for moving from

character to character rapidly. Line 335 handles toggling a bit, and redrawing the associated screen pixels. First, the current bit is determined by a GOSUB 450. Then the subroutine at line 470 changes the appropriate bit, and finally, the routine at line 400 is called to update the pixels in all the windows. All of these routines will come in handy in a minute when we discuss Invert and Clear. Finally, the routine restores the ON KBD condition, and returns to the "flashing cursor" loop in line 270.

That leaves only Invert and Clear as major, undiscussed functions. These are callable from Edit mode directly, or in command mode. Let's take Invert first, the more complicated of the two. Line 1500 prompts for clearing a whole row, a whole column, a block (defined as 8 bits wide, "ch" high), or the whole workspace. Rows are handled in lines 1540-1565 by moving through the row, subtracting 255 from each byte and storing them back. Then lines 1560-1565 set up and call 3610 to redraw the row. Inverting a row is handled by our bit-toggle routines, called repeatedly in lines 1575-1585 as if we were inverting each one separately with the space bar. Inverting a block and inverting the whole work space are handled in 1600-1655 as special cases of the invert row technique discussed above.

Once you understand the Invert techniques, Clear becomes simple, since it mostly involves zeroing out various locations. Note however, that in clearing a column in lines 1765-1775, that the bit toggle and draw (470 and 400) are only performed if the bit is on ("cstate" is true). Also, when the Clear workspace command is executed, line 600 is called to clear the windows fast, instead of drawing the pixels (all zeros) in them.

At Long Last, the program!

Well, there you have it, a monument to the Apple III graphics capability. Next month we will continue on with this topic, and use the editor to create shapes to populate our games and other graphic adventures. Until then, happy typing!

```
 5   REM Shape, Character and Font Editor
 10   DIM
char%(127,15),shape%(7,15),name$(10),ary$(10),size%(10),bits%(15,3)
 15   DIM
work%(7,15),shex%(15,3),cset%(511),lookup(15),flip(255),block$(15)

 20   PRINT"Initializing variables, please wait"
 25   GOSUB 4000
 30   INVOKE"/basic/bgraf.inv","/basic/request.inv","/basic/
download.inv"
 35   OPEN#1,".grafix"
 40   PERFORM initgrafix
 45   PERFORM grafixmode(%2,%1)
```

```
 50   PERFORM fillcolor(%15):PERFORM pencolor(%0)

 55   HOME:PRINT:PRINT"Initializing the graphics screen, please wait."
 60   PERFORM viewport(%0,%559,%0,%191):PERFORM fillport
 65   PERFORM moveto(%0,%184)
 70
PRINT#1;"============================================================
=
      ================";
 75   PERFORM moveto(%0,%191)
 80   PRINT#1 USING"79c";"DrawImage Editor"
 85
PRINT#1;"============================================================
=
      ================";
 90   PERFORM fillcolor(%0):PERFORM pencolor(%15)
 95   PERFORM moveto(%261,%176):PRINT#1;" Work Area "
 100   GOSUB 600
 125   PERFORM viewport(%5,%556,%13,%58):PERFORM fillport
 130   PERFORM viewport(%5,%556,%1,%10):PERFORM fillport
 135   PERFORM viewport(%0,%559,%0,%191)
 140   PERFORM moveto(%28,%128):PRINT#1;" 560 X 192 ";
 145   PERFORM moveto(%253,%128):PRINT#1;" 280 X 192 ";
 150   PERFORM moveto(%233,%98):PRINT#1;" 140 X 192 ";
 155   PERFORM moveto(%233,%67):PRINT#1;" Command Keys "
 165   PERFORM moveto(%7,%57):PRINT#1;"   Arrow keys move cursor
ESCAPE
quits
        current mode    SPACE toggles bits";
 170   PERFORM moveto(%69,%45):PRINT#1;" 0 : Catalog       3 : Delete
6
        : Clear"
 180   PERFORM moveto(%69,%36):PRINT#1;" 1 : Save          4 : Edit
7
        : Define"
 190   PERFORM moveto(%69,%27):PRINT#1;" 2 : Load          5 : Invert
8
        : View"
 200   PERFORM grafixon

 210   prompt$="Select a Command: "
 215   GOSUB 3000
 220   IF fin THEN 1000
 225   a=ASC(MID$(line$,1,1))
 230   IF a>47 AND a<57 THEN ON a-47 GOSUB
```

```
1200,1100,1400,1300,250,1500,1700,25
       00,1900:GOTO 210
 235   GOSUB 3070:GOTO 210

 250   GOSUB 450
 255   cflash= NOT cstate
 260   ON KBD GOTO 280
 265   PERFORM moveto(%chorz*4+7,%cvert*2+134)
 270   PERFORM drawimage(@shex%(0,0),%32,%24+cflash*4,%0,%4,%2)
 275   cflash= NOT cflash:FOR z=1 TO 5+200*( NOT cstate):NEXT:GOTO 270

 280   OFF KBD:PERFORM drawimage(@shex%(0,0),%v32%,
%24+cstate*4,%0,%4,%2)
 285   key= KBD:IF key=27 THEN kvl=0:POP:GOTO 210
 290   IF key>127 THEN skp=cw:key=key-128:ELSE:skp=1
 295   kvl=INSTR(ctrl$,CHR$(key))
 300   IF kvl THEN ON kvl GOTO 315,320,325,330,335,1500,1700
 305   ON KBD GOTO 280
 310   RETURN
 315   IF left<=chorz-skp THEN chorz=chorz-skp:GOSUB 450:GOTO
340:ELSE:GOTO
340
 320   IF right>=chorz+skp THEN chorz=chorz+skp:GOSUB 450:GOTO
340:ELSE:GOTO
34
       0
 325   IF top>=cvert+skp THEN cvert=cvert+skp:GOSUB 450:GOTO
340:ELSE:GOTO 340
 330   IF bot<=cvert-skp THEN cvert=cvert-skp:GOSUB 450:GOTO
340:ELSE:GOTO 340
 335   GOSUB 450:GOSUB 470:GOSUB 400:IF ch<15-cvert THEN ch=15-cvert
 340   PERFORM moveto(%chorz*4+7,%cvert*2+134)
 345   ON KBD GOTO 280
 350   RETURN

 400   PERFORM moveto(%chorz*4+7,%cvert*2+134):PERFORM
drawimage(@shex%(0,0),%3
       2,%24+cstate*4,%0,%4,%2)
 410   PERFORM moveto(%chorz+7,%cvert+102):PERFORM
drawimage(@shex%(0,0),%32,%6
       +cstate,%3,%1,%1)
 415   PERFORM moveto(%chorz*2+157,%cvert+102):PERFORM
drawimage(@shex%(0,0),%3
       2,%12+cstate*2,%2,%2,%1)
 420   PERFORM moveto(%chorz*4+7,%cvert+72):PERFORM
```

```
drawimage(@shex%(0,0),%32,%
       24+cstate*4,%0,%4,%1)
 425   RETURN

 450   col=INT(chorz/16):bitnum=chorz-col*16
 455
cval$=HEX$(work%(col,15-cvert)):nibpos=INT(bitnum/4):nib$=MID$(cval$,nib
       pos+1,1)
 460   bit=bitnum-nibpos*4:cstate=bits%(TEN(nib$),bit)
 465   RETURN

 470   cnval=2^(3-bit):IF cstate THEN cnval=-cnval
 475   SUB$(cval$,nibpos+1,1)=MID$(HEX$(TEN(nib$)+cnval),4,1)
 480   work%(col,15-cvert)=TEN(cval$)
 485   cstate= NOT cstate
 490   RETURN

 600   PERFORM viewport(%5,%556,%131,%166):PERFORM fillport
 605   PERFORM viewport(%5,%135,%101,%118):PERFORM fillport
 610   PERFORM viewport(%155,%420,%101,%118):PERFORM fillport
 615   PERFORM viewport(%5,%540,%71,%88):PERFORM fillport
 620   PERFORM viewport(%0,%559,%0,%191)
 625   RETURN

 1000   REM clean up and go home
 1005   HOME:TEXT
 1010   PERFORM release:PERFORM release:PERFORM release
 1015   INVOKE
 1020   CLOSE
 1030   END

 1100   IF choice=1 THEN filtyp=1:GOTO 1125
 1105   prompt$="Save as a "+name$(1)+", "+name$(2)+" or "+name$(3)+"? "
 1110   GOSUB 3000:IF fin THEN RETURN
 1115   a$=MID$(line$,1,1):filtyp=INT(INSTR("  Ss Cc Ff",a$)/3)
 1120   IF filtyp=0 THEN 1105
 1125   prompt$="Pathname of Save file: ":GOSUB 3000
 1130   IF fin AND choice=1 THEN RETURN:ELSE:IF fin THEN 1105
 1135   ON ERR GOTO 1190
 1140   OPEN#3,line$
 1145   IF TYP(3)=8 THEN message$="INVALID, "+line$+" is a TEXT
file.":GOSUB
31
       00:GOTO 1125
```

```
 1150   IF TYP(3)=0 THEN 1170
 1155   prompt$="Ok to destroy old data in file "+line$+"? ":GOSUB 3000
 1160   IF fin THEN 1125
 1165   IF NOT INSTR("Yy",MID$(line$,1,1)) THEN 1125
 1170   IF filtyp<>3 THEN WRITE#3,0;filtyp,ch,cw,sl:WRITE#3,1;0:READ#3,1
 1175   ON choice GOSUB 3750,3850,3860
 1178   IF choice>1 AND choice<>filtyp THEN GOSUB 2000
 1180   array$=ary$(filtyp):PERFORM filwrite(%3,@array$,%size%(filtyp))
 1185   message$=name$(filtyp)+" saved.":GOSUB 3100:CLOSE#3:OFF
ERR:RETURN
 1190   message$="Error in opening or writing to file. ":GOSUB 3100
 1195   OFF ERR:GOTO 1125

 1200   prompt$="Pathname to Catalog: "
 1205   GOSUB 3000
 1210   IF fin THEN delay=1:RETURN
 1215   oldpre$= PREFIX$
 1220   ON ERR GOTO 1270
 1225   PREFIX$=line$
 1230   OPEN#8 AS INPUT, PREFIX$
 1235   OFF ERR
 1240   ON EOF#8 GOTO 1285
 1245   delay=0
 1250   INPUT#8;message$
 1255   IF MID$(message$,1,10)="          " THEN 1250
 1260   GOSUB 3100
 1265   GET a$:IF ASC(a$)=27 THEN 1285:ELSE GOTO 1250
 1270   message$=line$+" is not a valid Prefix"
 1275   delay=1:GOSUB 3100
 1280   OFF ERR
 1285   PREFIX$=oldpre$
 1290   GOTO 1200

 1300   prompt$="Pathname of file to Delete: "
 1305   GOSUB 3000
 1310   IF fin THEN RETURN
 1315   ON ERR GOTO 1360
 1320   OPEN#8 AS INPUT,line$
 1325   IF TYP(8)<>1 THEN message$=line$+" is not a Save file":GOSUB
3100:CLOSE
        #8:GOTO 1300
 1330   ON ERR GOTO 1380
 1335   CLOSE#8:DELETE line$
 1340   OFF ERR
```

```
 1345   message$=line$+" deleted."
 1350   GOSUB 3100:GOTO 1300
 1360   OFF ERR
 1365   message$="Cannot delete "+line$+". (doesn't exist or can't be
opened)"
 1370   GOSUB 3100:GOTO 1300
 1380   OFF ERR
 1385   message$="Cannot delete "+line$+". (write-protected or locked)"
 1390   GOSUB 3100:GOTO 1300

 1400   prompt$="Load a "+name$(1)+", a "+name$(2)+" or a "+name$(3)+"?
"
 1405   GOSUB 3000:IF fin THEN RETURN
 1410   a$=MID$(line$,1,1):choice=INT(INSTR("  Ss Cc Ff",a$)/3)
 1415   IF choice<1 OR choice>3 THEN GOSUB 3070:GOTO 1400
 1420   prompt$="Pathname of "+name$(choice)+": "
 1425   GOSUB 3000:IF fin THEN 1400
 1430   ON ERR GOTO 1455
 1435   array$=ary$(choice):IF choice<>3 THEN OPEN#3,line$:GOTO 1450
 1440   ch=7:font$=CHR$(34)+line$+CHR$(34):PERFORM
getfont(@font$,@array$)
 1445   OFF ERR:GOSUB 3950:GOTO 1485
 1450   IF TYP(3)=1 THEN READ#3;filtyp,ch,cw,sl:IF filtyp=choice THEN
1470
 1455   message$="Not a "+name$(choice)+" file.":GOSUB 3100
 1460   OFF ERR:IF choice=3 THEN 1420
 1465   CLOSE#3:IF TYP(3)=0 THEN DELETE line$:GOTO 1420:ELSE:GOTO 1420
 1470   READ#3,1:PERFORM filread(%3,@array$,%size%(filtyp),@ret%)
 1475   CLOSE#3:IF ret%=size%(filtyp) THEN 1485
 1480   message$=name$(choice)+" in "+line$+" is invalid.":GOSUB
3100:GOTO
1420
 1485   GOSUB 3600:message$=name$(choice)+" loaded.":GOSUB 3100
 1490   RETURN

 1500   prompt$="Invert Row, Column, Block or Work space? "
 1505   GOSUB 3000:IF fin AND kvl THEN GOSUB 3500:GOSUB 450:GOTO 340
 1510   IF fin THEN RETURN
 1515   a$=MID$(line$,1,1)
 1520   a=INT(INSTR("  Rr Cc Bb Ww",a$)/3)
 1525   IF NOT a THEN GOSUB 3060:GOTO 1500
 1530   ON a GOTO 1540,1570,1600,1640
 1540   crow=15-cvert
 1545   FOR i=0 TO sl:b$=HEX$(work%(i,crow))
 1550
```

```
work%(i,crow)=TEN(MID$(HEX$(255-TEN(MID$(b$,1,2))),3,2)+MID$(HEX$(255
          -TEN(MID$(b$,3,2))),3,2))
 1555     NEXT
 1560   rs=crow:re=crow:bw=0:ew=sl
 1565   GOSUB 3610:GOTO 1500
 1570   cur.vert=cvert
 1575   FOR cvert=15-ch TO 15
 1580     GOSUB 450:GOSUB 470:GOSUB 400
 1585     NEXT
 1590   cvert=cur.vert
 1595   GOTO 1500
 1600   cloc=INT(chorz/16):chalf=(chorz-16*cloc>7):st=chalf*2+1
 1605   FOR i=0 TO ch:b$=HEX$(work%(cloc,i))
 1610     SUB$(b$,st,2)=MID$(HEX$(255-TEN(MID$(b$,st,2))),3,2)
 1615     work%(cloc,i)=TEN(b$)
 1620     NEXT i
 1625   bw=cloc:ew=cloc:rs=0:re=ch
 1630   GOSUB 3610:GOTO 1500
 1640   FOR crow=0 TO ch:FOR i=0 TO sl:b$=HEX$(work%(i,crow))
 1645
work%(i,crow)=TEN(MID$(HEX$(255-TEN(MID$(b$,1,2))),3,2)+MID$(HEX$(2
            55-TEN(MID$(b$,3,2))),3,2))
 1650       NEXT:NEXT
 1655   GOSUB 3607:GOTO 1500

 1700   prompt$="Clear Row, Column, Block or Work space? "
 1705   GOSUB 3000:IF fin AND kvl THEN GOSUB 3500:GOSUB 450:GOTO 340
 1710   IF fin THEN RETURN
 1715   a$=MID$(line$,1,1)
 1720   a=INT(INSTR("  Rr Cc Bb Ww",a$)/3)
 1725   IF NOT a THEN GOSUB 3060:GOTO 1700
 1730   ON a GOTO 1740,1760,1800,1830
 1740   crow=15-cvert
 1745   FOR i=0 TO sl:work%(i,crow)=0:NEXT
 1750   rs=crow:re=crow:bw=0:ew=sl
 1755   GOSUB 3610:GOTO 1700
 1760   cur.vert=cvert
 1765   FOR cvert=15-ch TO 15
 1770     GOSUB 450:IF cstate THEN GOSUB 470:GOSUB 400
 1775     NEXT
 1780   GOTO 1700
 1800   cloc=INT(chorz/16):chalf=(chorz-16*cloc>7):st=chalf*2+1
 1805   FOR i=0 TO ch:b$=HEX$(work%(cloc,i))
 1810     SUB$(b$,st,2)="00":work%(cloc,i)=TEN(b$):NEXT
```

```
 1815   bw=cloc:ew=cloc:rs=0:re=ch
 1820   GOSUB 3610:GOTO 1700
 1830   FOR crow=0 TO ch:FOR i=0 TO sl:work%(i,crow)=0:NEXT:NEXT
 1835   GOSUB 600:GOTO 1700

 1900   IF choice=1 THEN GOSUB 3605:RETURN
 1905   IF choice=2 THEN GOSUB 3850:ELSE:GOSUB 3860
 1910   GOSUB 3800
 1915   GOSUB 3605
 1920   RETURN

 2000   ON filtyp-1 GOTO 2100,2200

 2100   message$="Transferring Font format to Character set
format":GOSUB 3100
 2105   FOR k=0 TO 63:j=8*k-1
 2110     FOR i=0 TO 7 STEP 2:j=j+1:a$=HEX$(cset%(j)):b$=HEX$(cset%
(j+4))
 2115       char%(k,i)=TEN(MID$(a$,1,2)+MID$(b$,1,2))
 2120       char%(k,i+1)=TEN(MID$(a$,3,2)+MID$(b$,3,2))
 2125       NEXT:NEXT
 2130   FOR k=64 TO 127:FOR i=0 TO 7:char%(k,i)=0:NEXT:NEXT
 2135   FOR k=0 TO 127:FOR i=8 TO 15:char%(k,i)=0:NEXT:NEXT
 2140   RETURN

 2200   message$="Transferring Character set format to Font
format":GOSUB 3100
 2205   FOR k=0 TO 63:j=8*k-1
 2210     FOR i=0 TO 7 STEP 2:j=j+1:a$=HEX$(char%(k,i)):b$=HEX$(char%
(k,i+1))
 2215       cset%(j)=TEN(MID$(a$,1,2)+MID$(b$,1,2))
 2220       cset%(j+4)=TEN(MID$(a$,3,2)+MID$(b$,3,2))
 2225       NEXT:NEXT
 2230   GOSUB 3950
 2235   RETURN

 2500   prompt$="Character height is now "+CONV$(ch+1)+". New value: "
 2505   GOSUB 3000:IF fin THEN 2550
 2510   a=CONV(line$)
 2515   IF a<1 OR a>16 THEN message$="Character height must be between 1
and
16
        ":GOSUB 3100:GOTO 2500
 2520   ch=a-1:message$="Character height is now "+CONV$(ch+1)+".":GOSUB
3100
```

```
 2550   prompt$="Character width is now "+CONV$(cw)+". New value: "
 2555   GOSUB 3000:IF fin THEN 2600
 2560   a=CONV(line$)
 2565   IF a<1 OR a>255 THEN message$="Character width must be between 1
and
25
        5":GOSUB 3100:GOTO 2550
 2570   cw=a:message$="Character width is now "+CONV$(cw)+".":GOSUB 3100
 2600   prompt$="Work area width in dots (must be 16,32,48,64,80,96,112
or
128:
         "
 2605   GOSUB 3000:IF fin THEN 2700
 2610   a=CONV(line$)/16-1
 2615   IF INT(a)<>a THEN message$="Width must be a multiple of
16":GOSUB
3100:
        GOTO 2600
 2620   IF a<0 OR a>7 THEN message$="Width must be between 16 and
128":GOSUB
31
        00:GOTO 2600
 2625   sl=a:message$="Work area width is now "+CONV$((a+1)*16)+".:gosub
3100
 2630   right=(a+1)*16-1
 2635   message$="Definitions complete.":GOSUB 3100
 2640   RETURN

 3000   REM Accept a message from the window
 3005   GOSUB 3500:PERFORM moveto(%7,%9):PRINT#1;prompt$;
 3010   line$="":fin=0
 3015   GET a$:a=ASC(a$)
 3020   IF a>31 THEN PRINT#1;a$;:line$=line$+a$:GOTO 3015
 3025   IF a=13 THEN fin=LEN(line$)=0:RETURN
 3030   IF a=27 THEN fin=2:RETURN
 3035   IF a<>8 THEN 3015
 3040   IF LEN(line$)=0 THEN 3015
 3045   PERFORM moverel(%-7,%0):PRINT#1;" ";:PERFORM moverel(%-7,%0)
 3050   line$=MID$(line$,1,LEN(line$)-1)
 3055   GOTO 3015

 3060   REM print an error message
 3070   PERFORM moveto(%450,%9)
 3075   PRINT#1;"INVALID";:FOR i=1 TO 500:NEXT
 3080   PERFORM moveto(%450,%9)
```

```
 3085   PRINT#1;"       ";
 3090   RETURN

 3100   GOSUB 3500
 3110   PERFORM moveto(%7,%9):PRINT#1;message$;
 3120   FOR i=1 TO 750*delay:NEXT
 3130   RETURN

 3500   PERFORM viewport(%5,%556,%1,%10):PERFORM fillport
 3510   PERFORM viewport(%0,%559,%0,%191)
 3520   RETURN

 3600   ON choice GOSUB 3700,3800,3800
 3605   GOSUB 600
 3607   rs=0:re=ch:bw=0:ew=sl
 3610   xdot=7:ydot=164:rows=2:width=16:srow=0
 3615   GOSUB 3670
 3620   PERFORM moveto(%7,%117)
 3625   PERFORM drawimage(@work%(0,0),%16,%0,%0,%128,%ch+1)
 3630   xdot=157:ydot=117:rows=1:width=8:srow=2
 3635   GOSUB 3670
 3640   xdot=7:ydot=87:rows=1:width=16:srow=0
 3645   GOSUB 3670
 3650   RETURN

 3670   xhorz=xdot+16*bw*(width/4)
 3675   FOR k=rs TO re:PERFORM moveto(%xhorz,%ydot-rows*k):FOR i=bw TO
ew
 3680       a$=HEX$(work%(i,k)):FOR j=1 TO 4:dhex%=TEN(MID$
(a$,j,1))*width
 3685         PERFORM drawimage(@shex%(0,0),%32,%dhex%,%srow,%width,
%rows)
 3690         PERFORM moverel(%width,%0):NEXT:NEXT:NEXT
 3695   RETURN

 3700   FOR i=0 TO 7:FOR j=0 TO ch:work%(i,j)=shape%(i,j):NEXT:NEXT
 3705   RETURN

 3750   FOR i=0 TO 7:FOR j=0 TO 15:shape%(i,j)=work%(i,j):NEXT:NEXT
 3755   RETURN

 3800   prompt$="Starting Character number to display: "
 3805   GOSUB 3000
 3810   IF fin THEN cr=0:GOTO 3850
 3815   cr=VAL(line$)
```

```
 3820   IF cr<0 OR cr>254 THEN message$="Number out of range":GOSUB
3100:GOTO 3800
 3822   IF choice=3 THEN 3900
 3825   IF cr/2<>INT(cr/2) THEN message$="Character number must be even
(0,2,4,etc.)":GOSUB 3100:GOTO 3800
 3830   wd=cr/2:sl=7:IF wd+sl>127 THEN sl=127-wd
 3835   FOR i=0 TO ch:FOR j=0 TO sl:work%(j,i)=char%(wd+j,i):NEXT:NEXT
 3840   RETURN

 3850   FOR i=0 TO ch:FOR j=0 TO sl:char%(wd+j,i)=work%(j,i):NEXT:NEXT
 3855   RETURN

 3860   FOR k=0 TO sl:j=skip+8*k-1
 3865     FOR i=0 TO 7 STEP 2:j=j+1:a$=HEX$(work%(k,i)):b$=HEX$(work%
(k,i+1))
 3870       cset%(j)=TEN(MID$(a$,1,2)+MID$(b$,1,2))
 3875       cset%(j+4)=TEN(MID$(a$,3,2)+MID$(b$,3,2))
 3880       NEXT:NEXT
 3885   RETURN

 3900   IF cr>127 THEN message$="Character range must be 0-127":GOSUB
3100:GOTO 3800
 3910   skip=4*cr:sl=7:IF cr+2*sl>126 THEN sl=(126-cr)/2
 3915   FOR k=0 TO sl:j=skip+8*k-1
 3920     FOR i=0 TO 7 STEP 2:j=j+1:a$=HEX$(cset%(j)):b$=HEX$(cset%
(j+4))
 3925       work%(k,i)=TEN(MID$(a$,1,2)+MID$(b$,1,2))
 3930       work%(k,i+1)=TEN(MID$(a$,3,2)+MID$(b$,3,2))
 3935       NEXT:NEXT
 3940   RETURN

 3950   message$="Preparing the character font.":GOSUB 3100
 3955   FOR k=0 TO 511:b$=HEX$(cset%(k)):cset%(k)=TEN(HEX$
(v256*flip(TEN(MID$(b$,1,2)))+flip(TEN(MID$(b$,3,2))))):NEXT
 3960   RETURN

 4000   DATA 0000,000F,00F0,00FF,0F00,0F0F,0FF0,0FFF
 4005   DATA F000,F00F,F0F0,F0FF,FF00,FF0F,FFF0,FFFF
 4010   DATA 0003,0C0F,3033,3C3F,C0C3,CCCF,F0F3,FCFF
 4015   DATA 0123,4567,89AB,CDEF
 4025   DATA 0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15
 4026   DATA
0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,1,0,0,0,1,0,1,0,1,1,0,0,1,1,1
 4027   DATA
1,0,0,0,1,0,0,1,1,0,1,0,1,0,1,1,1,1,0,0,1,1,0,1,1,1,1,0,1,1,1,1
```

```
 4028   FOR i=0 TO 15:READ block$(i):NEXT
 4030   FOR i=0 TO 15:h%=TEN(block$(i)):shex%(i,0)=h%:shex%(i,1)=h%:NEXT
 4035   FOR i=0 TO 7:READ a$:shex%(i,2)=TEN(a$):NEXT
 4040   FOR i=0 TO 3:READ a$:shex%(i,3)=TEN(a$):NEXT
 4045   FOR i=0 TO 15:READ lookup(i):NEXT
 4050   FOR i=0 TO 15:FOR j=0 TO 3:READ bits%(i,j):NEXT:NEXT
 4055   v256=256:v16=16
 4060   FOR i=0 TO 255:a$=HEX$(i):flip(i)=v16*lookup(TEN(MID$(a$,4,1)))
+lookup(TEN(MID$(a$,3,1))):NEXT
 4065   sh=7:sl=7:ch=7:cw=8:choice=2:cr=0:wd=0:skip=0
 4070   name$(1)="Shape definition":name$(2)="Character
set":name$(3)="Font"
 4075   ary$(1)="shape%":ary$(2)="char%":ary$(3)="cset%"
 4080   size%(1)=256:size%(2)=2048:size%(3)=1024
 4085   ctrl$=CHR$(8)+CHR$(21)+CHR$(11)+CHR$(10)+CHR$(32)+"5"+"6"
 4090   left=0:right=127:top=15:bot=0:cvert=15:chorz=0:delay=1
 4500   RETURN
```

Exploring Business Basic, Part XVIII

Looking Through a Glass Backward

Last month's article was a lengthy tome on editing character fonts and shapes, delivered under the promise that it could be used to create "fun stuff". What else excuses the need to type in such a long program? That program, or something equivalent, is going to become pretty important this time, as we continue our discussions of graphics by developing an arcade "Shoot-em-up" called "Bug-Mania". In the process, we'll explore an area of graphics that most people overlook, the 40 column color-on-color text mode.

Now that we can create special character shapes and load them into the standard character set, a class of graphics known as "character set animation" is not only possible, but highly practical and rewarding. Most people don't realize it, but the capabilities of the Apple]["DOS Toolkit" Animatrix package are supplied as standard in the Apple III, and work at hardware speeds! However, to make all this neat stuff really work for you, you need a character set editor, such as the one discussed last time. If you don't have last month's article, you can still use the programs, but they won't make much sense, and they certainly won't be pretty.

Since last month's article was written, a set of Business Basic Invokable Modules has appeared on the market from Foxware Products of Salt Lake City, Utah. I highly recommend that you get a copy of this product, which they call "BASIC Extension". In addition to routines to manipulate and search arrays, it contains useful functions like Reset lockout, reboot, upshift, and, most interesting to last month's article, the ability to change file types. Sepcifically, the editor from the last article has been modified with this routine to automatically change the edited font file to the system filetype "FONT", so that it can be used by other system proceedures. This allows you to edit a font and then configure it with SCP, etc. Congratulations, Foxware!

Doing the Sideways Scroll

One of the critical elements of any "shoot-em-up" game is the ability to keep more than one activity going on the screen simultaneously. This means that the targets should move along with the weapons, and firing at objects should not bring everything to a halt. Since each of these activities takes some processing time, it is important that each take as little time as possible, in order to make the whole game smooth and fast. Normally on the Apple][and other similar systems, this is accomplished with assembly language routines, which execute fast enough to make up for the fact that it takes a lot of work to move objects

around on the hi-res screen. Careful use of the Apple III built-in routines, and the user-definable character set will allow us to do much the same thing in Basic! To begin, let's look at a routine which scrolls objects horizontally across the text screen. We'll start with using groups of asterisks (realizing that we could redefine them to almost any shape).

The program looks like this:

```
 10   m$=" **  ****  **  ****  *** * ****   ** ***"
 20   start$=CHR$(26)+CHR$(0)+CHR$(10)
 30   PRINT CHR$(21);"1";
 40   PRINT CHR$(16);CHR$(1);
 50   PRINT CHR$(19);CHR$(4);
 60   PRINT CHR$(20);CHR$(13);
 70   HOME
 80   FOR i=0 TO 39
 90     PRINT start$;MID$(m$,41-i,i);MID$(m$,1,40-i)
100     GET a$:IF ASC(a$)=27 THEN 130
110     NEXT i
120   GOTO 80
130   TEXT:HOME
140   END
```

Line 10 defines a 40 character string, exactly as wide as the screen mode we will use. Be sure when you type it in that the result is exactly 40 characters in length. Line 20 puts a starting location to display the strings on the screen into "start$", using the cursor positioning command of the console driver. The code shown will set a starting location at column 0, row 10. Line 30 sets the cursor movement options to disable everything but "advance after character". You should check out the Standard Device Drivers Manual for more details on the cursor options of the Console Driver. Line 40 puts the screen into 40 column color text mode, and lines 50 and 60 set foreground color to dark green (4) and the background color to yellow (13). The HOME command sets the whole screen to the background color, and then the routine to scroll m$ begins in line 80.

It is important to go through the routine at lines 80 through 110 to see how this accomplishes the display of the string "m$". For each value of "i", the string is split into two parts, on the boundary between the beginning and end of the string. When i=0, then the result is "MID$(m$,41,0);MID$(m$,1,40)". This is the combination of the "null" string, together with the entire string (1,40). For i=1, the combination is "MID$(m$,40,1);MID$(m$,1,39)", that is, the last character of the string, coupled with the first 39 characters. The end of the sequence, i=39, is equivalent to "MID$(m$,2,39);MID$(m$,1,1)" and finishes rotating the string around to start again. The GET statement in line 100 allows the display to freeze after each step, so you can see exactly how this works. To see the effect of the scrolling, simply hold down any key (except ESCAPE) and the pattern will scroll rapidly to the right. Holding down the "closed-Apple" key in combination with

the other key will cause faster scrolling (since the characters are presented to GET faster). If you want to see how fast this thing will really run, cut out line 100 completely. Those asterisks will really fly!

Line 130 is important since it snaps you back to 80 column reality with the cursor options restored.

Well, so much for the simple stuff. Now it's time to add the options to create objects and creatures of your own to populate your game world.

"Oh Scroll a Mio"

We'll start by defining the characters which will bring our creature to life. To make things interesting, we'll use two versions of the "bug", so that we can produce some simple animation without getting too complicated. Arbitrarily the characters from decimal 20 through 25 are picked to be redefined. These normally are "Control characters" and are not displayed unless referenced by their character number + 128, that is, decimal 148 through 153. This prevents the animation characters from interfering with any other normal character printing we may have to do. If you haven't seen the last issue, and don't have access to a character set editor, you may simply use the same character numbers without changing their definition, but the "creatures" will not make any sense.

In any case, here's a suggestion for the "bug" set:

```
   20 (148)          21 (149)          22 (150)

_______________   _______________   _______________
|_|_|_|_|_|_|_|   | | |x|x|x| |_|   |_| |x|x|x| |_|
|_|_| |_|_|_| |   |x|x|x|x|x|x| |   | |x|x|x| |x| |
|_| |x|_|_| |x|   |x|x| |x| |x|x|   |x|x|x|x|x|x|x|
|_|x|_|_| |x|x|   |x|x|x|x|x|x|x|   |x|x|_|_|_|x|_|
|_|x| | |x|_|x|   |x|x|x|x|x|x|x|   |x|x| |_|_|_| |
|_|_|x|x|_|_|_|   |_|_|x|_|_|x|_|   |_|_|x| | | |x|
|_|_|_|_|_|_|_|   |_|_|x| |_|x| |   |_|_|_|x|x|x|_|
|_|_|_|_|_|_|_|   |_|_|x|x|_|x|x|   |_|_|_|_|_|_|_|
   23 (151)          24 (152)          25 (153)

_______________   _______________   _______________
|_|_|_|_|_|_|_|   | | |x|x|x| |_|   |_| |x|x|x| |_|
|_|_|_| | |_| |   |x|x|x|x|x|x| |   | |x|x|x| |x| |
| |_| |x|x| |x|   |x|x|x| |x| |x|   |x|x| |x|x|x|x|
|x|_|x|_|_|x|x|   |x|x|x|x|x|x|x|   |x|x| | | |x|x|
|x| |x|_|_|_|x|   |x|x|x|x|x|x|x|   |x|x|x|x|x|x|_|
|_|x|_|_|_|_|_|   | | |x|_| |x|_|   |_|_|x|x|x|_|_|
|_|_|_|_|_|_|_|   |_|x|_|_|x|_|_|   |_|_|_|_|_|_|_|
|_|_|_|_|_|_|_|   |_|x|_|_|x|_|_|   |_|_|_|_|_|_|_|
```

Creepy, right? Actually, you can probably create a better looking "bug" than this, so play around with the editor until you are satisfied. Watch the "280 X 192" window on the editor to get an idea of how your creature will look in 40 column mode. Notice also that the changes from the set in 20-22 to the set in 23-25 are designed to make the mouth open and close, the legs move, and the tail wag! This is the origin of the concept "character set animation", since the animation of an object is accomplished by displaying several related versions of the object rapidly on the screen as characters.

Since the shapes will be displayed as a character font, remember the rules for system character fonts: Use the first seven dot positions only (the eighth is used for "flash/no flash" in inverse mode and will not be displayed) and make sure the characters are no more than eight dots high. Save the character set using the "font" option, with any name you like. If you have a way to change the resulting file to the official "FONT" type (via the invokable module discussed earlier, or the Pascal System Filer), do so now. This will save some hassles later.

Can't Tell One Bug from Another Without a Program

Now for a program which will display these characters on the screen and accomplish the animation! We'll use the scroll technique from the last program, together with character strings made up of our new character font.

```
10   DIM a%(511),char$(3)
15   q$=CHR$(34):esc$=CHR$(27):slen=40
20   array$="a%":char$(0)=" "
25   text40$=CHR$(16)+CHR$(0)
30   b$=" ":b2$="  ":b3$="   "

35   INVOKE"/BASIC/download.inv"
40   INPUT"Name of font file: ";flname$
45   name$=q$+flname$+q$
50   INPUT"Line number to crawl on: ";l
55   tc$=CHR$(26)+CHR$(0)+CHR$(l)
```

The lines above do the initalization of several arrays and values. Note that the array "a%" is dimensioned to hold an entire character set which "Download.inv" will load off disk. You'll have to change the pathname of "Download.inv" to be correct on your own system. In addition, "text40$" contains the console commands to turn on 40 column black and white mode (mode 0), and "tc$" contains the cursor addressing command to position the cursor to row 0, line "l", using the line number that was input on line 50.

```
60   char$(1)=CHR$(148):char$(2)=CHR$(149):char$(3)=CHR$(150)
65   m$=".23...123.1223..13.123.3.23...123.1223.."
```

```
 70   FOR i=1 TO 40:SUB$(m$,i,1)=char$(VAL(MID$(m$,i,1))):NEXT i
 75   char$(1)=CHR$(151):char$(2)=CHR$(152):char$(3)=CHR$(153)
 80   n$=".23...123.1223..13.123.3.23...123.1223.."
 85   FOR i=1 TO 40:SUB$(n$,i,1)=char$(VAL(MID$(n$,i,1))):NEXT i
```

Lines 60 through 85 look complicated, but they are simply the instructions on how to set up the strings to be scrolled across the screen. First, line 60 creates values in the "char$" array which correspond to the pieces of our first "bug". Then the "m$" string in line 65 tells what piece to put in what position. This allows us to create bugs which consist of a head only, a head and a tail, a head, body and tail, or any combination our imagination permits. The periods (".") in between the numbers are simply placeholders, which have a value of 0. Remember that we assigned a space to "char$(0)". Line 70 reads the values in "m$", one character at a time, and substitutes the appropriate value from the "char$" array. Then lines 75-85 do the same thing for a second string, "n$" which will contain the shifted versions of our bugs. Although we kept the bug structures the same in m$ and n$, nothing prevents us from redefining even the length of the shapes from one string to the next. This would allow bugs like caterpillars, which move by shortening and lengthening their bodies!

```
 90   PERFORM getfont(@name$,@array$)
 95   PRINT text40$;:HOME
 100   PERFORM loadfont(@array$)
 105   PRINT CHR$(21);"1";
```

Lines 90 through 105 get the font specified earlier, set up the screen mode, load the font into the standard character set, and turn off all screen options except advance, allowing us to write to the screen without interference from scrolling, etc.

NOTE: If you used the font editor from the last issue, or some other font editing technique, and cannot change the saved file as an "official" system type "FONT" (as shown by the CATalog listing), you must make some modifications to what's been described so far. The changes are:

```
35   INVOKE "/BASIC/download.inv","/BASIC/request.inv"
90   OPEN#1,name$:PERFORM filread(%1,@array$,%1024,@ret%)
92   IF ret%<>1024 THEN PRINT "Not a font file":GOTO 40
```

Remember, make these changes if you use the font editor from the last article and cannot change the saved file type to "FONT". And now, on with the show...

```
 110   FOR i=0 TO 39 STEP 2
 115     PRINT tc$;MID$(m$,slen+1-i,i);MID$(m$,1,slen-i)
 120     GET z$:IF z$=esc$ THEN 200
 125     PRINT tc$;MID$(n$,slen-i,i+1);MID$(n$,1,slen-i-1)
 130     GET z$:IF z$=esc$ THEN 200
 135     NEXT i
 140   GOTO 110
```

Lines 110-140 are the main scrolling loop. As you can see, it looks basicly (heh-heh) like the lines in the last program, with several important differences. Since we have two strings to print, we cut the number of interations in half, and adjust the subscripts in each MID$ function to print successive strings in the sequence. We still use the GET statement to pause between each change, but now it permits us to see the animation as it progresses. Again, holding down any key will permit smooth scrolling and motion as the little creatures open and shut their mouths, move their legs and wag their tails.

```
200   PRINT CHR$(21);"=";
205   PRINT CHR$(22);CHR$(14);
210   TEXT:HOME
215   name$=q$+"/BASIC/standard"+q$
220   PERFORM getfont(@name$,@array$):PERFORM loadfont(@array$)
225   PRINT CHR$(15);
230   END
```

Lines 200-230 perform cleanup, but in this case there's more to clean up than before. After setting console options back to normal (line 200), line 205 shuts off the screen while the cleanup is being done. The CHR$(22) is there to syncronize shutting off the screen with vertical blanking, to avoid funny flashes on the screen. Line 210 restores the 80 column screen and clears it to blanks, and then lines 215 to 225 restore the standard character set and turn the screen back on. You should change the pathname to the name of the character set you normally use.

Well, that wraps up the example program. When you run it, the creepy creatures should crawl across the screen at your command. Its fun to elaborate on this program, by editing more complex characters, or creating more versions of them to get smoother animation. In fact, the "Running Horse" demo on the System Demo disk was done somewhat in this way.

Business BASIC Gets a Little Gamey

By now we've covered all the essentials necessary for you to quickly follow the discussion of the "arcade" type game below. Basically, we're going to take our scrolling creatures and make them targets in a shooting gallery ("Oh, no" you cry, "not our poor creatures!"). To be a little fairer, we'll put some moving obstacles between the shooter and the creatures, and deduct points when the bullets hit the obstacles. Both the creatures and the obstacles will be moving using the techniques from the previous program. In addition, every good game needs some sound effects. We'll use the ".audio" driver to make some tones to liven up the game. With that said, lets look at the game:

```
5   DIM a%(511),dq$(39),eq$(39),fq$(39),lin$(3),blk$(3),j(255),pnts(4)
10   DIM m(40),char$(3),beep$(3)
```

```
 15   INVOKE".d3/download.inv"

 17   OPEN#1,".audio"
 20   bell$=CHR$(7):bp$=CHR$(128)+CHR$(63):ep$=CHR$(1)+CHR$(0)
 22   beep$(0)=bp$+CHR$(7)+CHR$(4)+ep$:beep$(1)=bp$+CHR$(8)+CHR$(6)+ep$
 23   beep$(2)=bp$+CHR$(18)+CHR$(5)+ep$:beep$(3)=bp$
+CHR$(197)+CHR$(6)+ep$
```

These lines set up the arrays to be used and create the tones which will indicate different kinds of hits. To understand lines 20 through 23 better, you should read the section of the Standard Device Drivers Manual on the programs.

```
 25   q$=CHR$(34):array$="a%":b$=" ":b2$="  ":b3$="   ":char$(0)=" "
 30   fg$=CHR$(19):bg$=CHR$(20):slen=40:na$=CHR$(21)+"0":
av$=CHR$(21)+"1"
 35   orange$=CHR$(9):green$=CHR$(12):mblue$=CHR$(6):white$=CHR$(15)
 40   og$=fg$+orange$+bg$+green$:bw$=fg$+mblue$+bg$+white$
 45   text40$=CHR$(16)+CHR$(1):t$=CHR$(26):t1$=t$+CHR$(0):bu$=CHR$(11)
 50   tc$=t1$+CHR$(4):t2$=t1$+CHR$(6):t3$=t1$+CHR$(8)
 55   t5$=t$+CHR$(8)+CHR$(23):t7$=t$+CHR$(35)+CHR$(23)
```

Lines 25-55 set up more constants for the program, especially the values for various foreground-background color combinations. This time we'll be using the 40 column color-on-color mode for more visual excitement. If you have a black and white (or black and green), the result will be shades of gray (or green). The strings "tc$", "t2$", "t3$", "t5$" and "t7$" will be used later on to position various other strings on the screen.

```
 60   l$="|"+bu$:l4$=l$+l$+l$+l$:l12$=l4$+l4$+l4$
 65   e$=" "+bu$:e4$=e$+e$+e$+e$:e12$=e4$+e4$+e4$
 70   lin$(0)=na$+l12$+l4$+"*":lin$(1)=na$+l12$:lin$(2)=na$+l12$+l$+l$:
      lin$(3)=lin$(1)
 75   blk$(0)=e12$+e4$+av$:blk$(1)=e12$+av$:blk$(2)=e12$+e$+e$+av$:
      blk$(3)=blk$(1)
 80   j(32)=1:j(8)=2:j(21)=3:j(13)=4:j(141)=5:j(27)=6
```

Lines 60-80 set up additional variables and strings needed for the program. In particular, the "lin$" array contains various versions of the characters used to represent firing of a shot at the creatures. It is made up of sets of vertical bars (the "|" character) combined with the vertical tab character contained in the "bu$" string. Vertical tab is used because the shot is fired from the bottom of the screen toward the top. Each string in the array is prefixed by the "na$" string, containing the screen control codes to turn off "character advance". This comes in handy in printing vertically, since it is only necessary to go up after printing, not back up and then go up as would be true if "advance" was on. Anything that reduces the number of characters printed on the screen speeds up the action. Notice also that "lin$(0)" has an asterisk as the last character.

This represents a burst as the shot goes through the barrier and explodes. By adding some extra characters, you could make the line and the burst different colors. Line 75 defines "blk$", which erases a shot right after it's fired. This gives the impression of a quick blast from the gunner. Notice that "av$" is added to the end of each occurrence of "blk$" to turn advance back on.

Line 80 sets up the values for the routine that decodes keystrokes and decides what to do. More on that later.

```
90   e$="==         ==      ===  ==        === =  =="
95   f$="  ===      == =      == ===      ==      ==="
100   FOR x1=0 TO 39 STEP 2:
      dq$(x1/2)=MID$(e$,slen+1-x1,x1)+MID$(e$,1,slen-x1):
      eq$(x1/2)=MID$(e$,slen-x1,x1+1)+MID$(e$,1,slen-x1-1):NEXT
105   FOR x1=20 TO 39:j=x1-20:dq$(x1)=dq$(j):eq$(x1)=eq$(j):NEXT
110   FOR x1=0 TO 39:fq$(x1)=MID$(f$,x1+1,slen-x1)+MID$(f$,1,x1):NEXT
```

Lines 90 through 110 set up the scrolling barriers mentioned above. "E$" and "f$" can be any arrangement you like, but be sure to make them exactly 40 characters long. Notice that instead of waiting and performing the MID$ functions when the string is actually printed on the screen, they are done in this loop and the results stored in string arrays to be printed later. Since these strings won't change, this becomes a more efficient, and therefore faster, way to handle them. Line 110 handles this in a straight-forward way, but lines 100 and 105 create "dq$" and "eq$" in a more confusing way. Basically, the process is this: "dq$" and "eq$" contain every other occurance of "e$", the string to be scrolled. By printing them both successively, along with one occurance of "fq$" the result will be that the first set of barriers will appear to scroll twice as fast as the first. Try this out in a simpler program if it's hard to follow.

```
140   GOSUB 700:REM get font
150   hr=20:points=0:hits=0
160   pnts(1)=0:pnts(2)=0:pnts(3)=0:hit=0
170   pnts(1)=pnts(1)+4:pnts(2)=pnts(2)+2:pnts(3)=pnts(3)+6:hit=hit+5
175   t4$=CHR$(26)+CHR$(hr-2)+CHR$(21):t6$=CHR$(26)+CHR$(hr-1)+CHR$(20)
180   GOSUB 800:REM load up the bugs
185   GOSUB 600:REM set up screen
```

Lines 140-185 do the last of the setup and perpare to play the game (at last! at last!). We'll discuss the subroutines in a minute. First, note that line 150 defines the initial position of the gunner, "hr=20". "Hr" will contain the horizontal position of the gunner at all times. Points scored and number of hits are also set to zero. Line 160 clears the point value array, and sets the value of a hit to zero, and line 170 increments them by a standard amount. This allows us to up the points values in each round, or start over, by going to lines 150, 160 or 170. The values in the "pnts" array are to be subtracted for hitting various combinations of barriers, and "hit" is a multiplier for scoring a hit on various parts of a

creature. "T4$" contains the position of the gunner, and "t6$" is the position from which the firing of "lin$" takes place.

Now we're ready to look at the additional setup subroutines. The first, at line 700, loads the font of our choice. It looks very similar to the routine from last time:

```
700   INPUT"Name of font file: ";flname$
705   name$=q$+flname$+q$
710   PERFORM getfont(@name$,@array$)
715   RETURN
```

Again, if you can't change your edited font files to type FONT, make the changes in this routine which were suggested in the last program.

The routine at line 800 is used to set up the "bug" character strings. It is derived from the last program and looks like this:

```
800   temp$=".12...123.1234..12.123.1.12...123.1234.."
805   char$(1)=CHR$(148):char$(2)=CHR$(149):char$(3)=CHR$(150)
810   m$(0)=".23...123.1223..13.123.3.23...123.1223.."
820   FOR i=1 TO 40:SUB$(m$(0),i,1)=char$(VAL(MID$(m$(0),i,1))):NEXT i
830   char$(1)=CHR$(151):char$(2)=CHR$(152):char$(3)=CHR$(153)
840   m$(1)=".23...123.1223..13.123.3.23...123.1223.."
850   FOR i=1 TO 40:SUB$(m$(1),i,1)=char$(VAL(MID$(m$(1),i,1))):NEXT i
855   FOR i=1 TO 40:m(i)=VAL(MID$(temp$,i,1)):NEXT i
860   RETURN
```

The only real difference here is in line 800 and 855. These lines combine to create the array "m", which is used to score hits and quickly decide how long an individual creature is. The game uses the principle that if you hit a creature, you destroy everything from the point of impact back, but whatever's left in front of the hit keeps going. We'll see exactly how this is done later.

The last setup routine creates the screen and playing area.

```
600   PRINT CHR$(14);
605   PRINT text40$;bw$;:HOME
610   PERFORM loadfont(@array$)
615   PRINT av$;:REM turn everything off but advance
630   PRINT og$;:PRINT USING"40c";b$
635   PRINT USING"40c";"Bug-Mania":PRINT USING"40c";b$
645   VPOS=23:PRINT USING"40c";b$:PRINT USING"40c";b$
650   PRINT"  Score:";:HPOS=31:PRINT"Hits:";
655   PRINT t5$;points;"    ";t7$;hits;
660   PRINT bw$;t4$;" X ";
665   PRINT CHR$(15);
670   RETURN
```

First the screen is turned off, so the setup can be completed quickly and without being seen. Then 40 column color mode (blue on white) is chosen and the screen is cleared to white (line 605). The font is then loaded as the standard character set, and various parts of the screen are filled in using orange on green color text. Then line 660 turns blue on white mode back on, and an "X" is printed at the current location of the gunner. If you want, you can edit the "X" character to any shape you like. Then the screen is turned back on in line 665 and the game is ready to play.

Getting Underway in Bugland

All the proceeding gets us ready for the actual playing proceedures. To make the game fun, we want to have scrolling and motion taking place all at once. As we have seen in previous articles, the best way to tackle that is the "ON KBD" capability, where the Apple III can be doing something but still respond when a key is pressed.

```
190   ON KBD GOTO 300

200   FOR i=0 TO 39:c=(i/2=INT(i/2)):
      g=(MID$(fq$(i),hr,1)<>b$)+(MID$(eq$(i),hr,1)<>b$)*2:
      PRINT tc$;MID$(m$(c),slen+1-i,i);MID$(m$(c),1,slen-i);
      t2$;dq$(i);t3$;fq$(i);t2$;eq$(i):NEXT:GOTO 200
```

Line 190 sets up an "ON KBD" jump to 300 when a key is pressed. In the meantime, line 200 is continously executed. This is a complicated routine, so let's look at each part of the line. First, the scrolling loop is set up as before. Then a value for "c" is calculated. "C" will be 0 if "i" is odd, 1 if "i" is even. This allows us to choose which version of the creatures we will display on this round. Then "g" is calculated. "G" is a number which indicated what the state of the barriers is. When you analyse the complicated logical expression, you will find that "g" is 0 if both "fq$" and "eq$" are blank at the current location of "hr" (the gunner's position). This would indicate that the barriers are open at that spot. "G" is equal to 1 if the bottom barrier ("fq$") is closed, and 2 if the top barrier ("eq$") is closed. If both barriers are closed, then "g" is 3. This will affect scoring, as we'll see later. Note that we really only need to calculate "g" when a shot is taken, but we've got time to waste in this loop, and we will need every millisecond when a shot is actually fired to avoid slowing down the game.

Next the appropriate version of the creature string is printed, and then the barrier strings are printed. Notice that printing "dq$", "fq$" and "eq$" in that order has the effect of scrolling the top barrier twice as fast as the creatures, in the same direction, and scrolling the bottom barrier at the same speed, except, because of the way "fq$" was created, it appears to scroll backwards. After scrolling through the entire "m$" array, the routine goes back and starts over endlessly, until a key is pressed.

Which brings us the the "ON KBD" routine at line 300:

```
 300   OFF KBD:z= KBD
 305   ON j(z) GOTO 400,330,340,150,170,500
 310   ON KBD GOTO 300
 315   RETURN
```

First we turn off the keyboard interrupt, and assign "z" the ASCII value of the character that was typed. This is used in line 305 to determine which processing routine to jump to. Check the definition of the "j" array in line 80 for more information about what is happening. This technique of branching is very wasteful of space (the "j" array takes up 1K of memory), but is extremely fast, which is what we need in processing these keystrokes. Cross referencing line 80 tells us that the jump to line 400 happens if z=32 (space bar). This is the firing signal. Lines 330 and 340 process right and left arrow keys (ASCII 8 and 21), respectively, which are used to move the gunner around. A RETURN (ASCII 13) jumps back to line 150 to begin a new game, and an "Open-Apple" RETURN (ASCII 141) restarts with doubled point and penalty values. Finally, an Escape (ASCII 27) jumps to 500 and ends the game. Looking at these individual routines will end our discussion of this game and get us down to playing it. Let's go:

```
 330   hr=hr-(hr>2):GOTO 350
 340   hr=hr+(hr<39)
 350   SUB$(t4$,2,1)=CHR$(hr-2):SUB$(t6$,2,1)=CHR$(hr-1):PRINT t4$;" X "
 360   ON KBD GOTO 300
 370   RETURN
```

This simply resets the value of "hr" after being sure that "hr" is not already at the left or right edge. You could reduce these values to restrict the gunner to a certain section of the screen. Line 350 changes the values of "t4$" and "t6$" to represent the new value of "hr" and reprints the gunner with spaces on each side. Printing the spaces erases the previous image of the gunner in the old position, no matter which way he moved. Then the ON KBD statement is re-activated, and the routine returns the the loop at 200.

```
 400   IF i=40 THEN 460
 405   PRINT t6$;lin$(g);t6$;blk$(g)
 410   IF g THEN points=points-pnts(g):PRINT#1;beep$(g):GOTO 450
 415   ch=slen*(i>hr)+hr-i
 420   IF NOT m(ch) THEN PRINT#1;beep$(g):GOTO 450:ELSE:PRINT bell$;
 425   FOR j=ch TO ch-m(ch)+1 STEP-1:IF m(j) THEN
points=points+m(j)*hit:m(j)=0:NEXT
 430   SUB$(m$(0),j+1,ch-j)="    ":SUB$(m$(1),j+1,ch-j)="
":hits=hits+1
 450   PRINT og$;t5$;points;"    ";t7$;hits;bw$
 460   ON KBD GOTO 300
 470   RETURN
```

Lines 400-470 are the firing subroutine, and this is where all the action takes place. First, a check is made to see if the keystroke happened during the loop exit and restart time. If so, a return is made with no action taken. This rarely happens, but must be provided for. Next, the gun is fired, by printing "lin$" and "blk$" at the current gunner location. Then "g" is checked to see if the bullet struck a barrier. If so, the appropriate number of points is deducted, and a tone is sounded with pitch corresponding to which barrier was struck. Then a jump is done to 450 to print the new point values and return.

If g=0, then the bullet made it through the barriers and a check is made in line 415 and 420 to see if anything was hit. If the "m" array contains 0 at that point, then the shot was a miss, the appropriate tone is sounded, and a return is made. If "m" is not zero, then a machine "bell" is sounded (note that a "BELL" character sounds without slowing down the program like a tone does). Line 425 then backs up along the string adding up points and zeroing out the "m" array. Line 430 blanks out the appropriate parts of the "m$" strings and bumps the hit count. With the strings changed, the next printing of "m$" will erase the bug from the point of the hit backwards. Line 450 then prints the new score and play resumes.

Notice that the major work of the game is done in this routine. Anything which makes this routine simpler or faster has the effect of speeding up play, and making the game more fun.

This finally brings us to the last routine, which ends the game:

```
500   PRINT CHR$(21);"="
505   PRINT CHR$(22);CHR$(14);
510   TEXT:HOME
515   nam$=q$+"/BASIC/standard"+q$
520   PERFORM getfont(@nam$,@array$):PERFORM loadfont(@array$)
525   PRINT CHR$(15);
530   CLOSE:INVOKE
540   END
```

This is essentially identical to the routine used in the previous program, and won't be elaborated on.

A Game a Day keeps Pac-man Away

Sure, this game won't save you many quarters if you're an Arcade freak, and it's not exactly going to drive Bill Budge out of business, but hopefully the techniques will prove useful, and prove something else as well. You don't need assembly language to get reasonable performance out of the Apple III, even in the realm of programming generally thought to absolutely require it, games. If you're careful, use clever techniques, and remember that you can trade off memory space for tables, etc. for additional speed, then you can create some

interesting things. There is certainly a lot you can do to make this month's game more interesting too. Try to figure out how to have the "heads" dive down through the barrier and attack the gunner when their bodies have been shot away. You might also try to speed up the scrolling by just printing one third of the gun blast at a time, with scrolling in between, and then figure the hit out at the end of the process. This would look more natural, and require the gunner to "lead" the target, quite a challenge. You can also change the scoring rules to your liking, and of course, completely redefine the barriers and bug shapes. Have fun!

Next month we'll add one important capability to this game, that of smooth scrolling with the character download capability. That's the real way the "Horse Demo" works. In addition, we'll start our exploration of how these techniques, and some brand new ones not possible in "text" mode, can be implemented on the high-resolution graphics screen. Until then, blaze away!

Exploring Business Basic, Part XIX

Here's hoping all of you got a chance to see the March article in this series, 'cause there's going to be lots of information in this month's missive which will build on that exploration of character graphics and the associated game which was dubbed "Bug-Mania". As was said last time, Business Basic wasn't exactly designed for games (the name is one clue!), but with careful design, reasonably performing graphics is possible, and that has a lot of business applications. In fact, the concept of redefining characters to new shapes can add tremendously to the effectiveness and readablity of your screen displays. For proof of this statement, take a look at Apple's Mail List Manager program some time. Special character sets in that program make it one of the best appearing and easiest to use applications on the Apple III.

But First, A Word from our Sponsors

The mailbag this month brought a couple of items of general interest. First, the question came up as to why Business Basic does not allow programs larger than 64K bytes. That is, the BASIC program statements cannot total more than 64K, excluding variable and array space, etc. This question rarely comes up, because 64K is enough space for over 3000 program lines, which is a very impractical size for a single program. One of these articles someday will cover all the tricks you can do with the "CHAIN" statement to break your program up into logical segments. But, the question deserves an answer, and in doing so, we'll note some other limits of Business Basic as well.

One of the reasons that limits like 64K keep coming up relates to the fact that a 16 bit pointer can store all the possible addresses in a 64K address space (2^15-1=65535). The Apple III can address more than 64K because it uses "extended indirect addressing", which makes use of "3-byte", 24 bit pointers. 3-byte pointers take up more space, however, and therefore are only used when it makes sense. The designers of Business Basic built in several such limits to save space and improve performance. For example, you are only allowed to have 64K of string variable space and 64K of simple numeric variable space. You are allowed as many numeric arrays as will fit, but each one must be no more than 64K. These may sound like limits, but remember, most personal computer Basics have an absolute limit of 64K of total space, program, data, the works. Ask your friends with an IBM PC what they get when they type FRE - right, about 61K no matter how much memory (above 128K) is actually installed!

It would be remiss to not note another, though more esoteric reason for the program size limit. Since Business Basic saves and loads programs with a single SOS call, and SOS has a 64K limit on the total number of bytes in a single

read or write, it would be impossible to use such a program, even if it could be written. So now you know.

The other question is one for which this columnist has no ready answer, so your comments are solicited. It concerns supporting the Qume letter quality printer as a graphics output device. The Apple III Business Graphics package supports the Qume, producing hi-res screen dumps and other graphic images, but those routines are written in Pascal. If anybody has developed Qume output routines, drop a note in care of Softalk and it will be passed along to the inquirer and to the rest of you in a later column.

Back to Work

Last time, as you may remember, we used the fact that the Apple III has a RAM-based, and therefore modifiable, character set to create some high speed animation effects. These were accomplished by redefining certain characters, and then printing them rapidly to the screen, taking advantage of the fact that printing to the text screen goes very rapidly, compared to writing to the graphics screen. Since the Apple III has a 16 color text mode, it can be tough to tell whether the action is occurring in text mode or graphics, especially if you do a lot of work with the character definitions.

There are, however, other ways to accomplish the rapid changing of characters on the screen for animation and other purposes. For example, suppose you wanted to change every occurrence of one character shape on the screen to another shape. You could simply re-print the character, and rely on the fact that printing in text mode is pretty fast, or you could take advantage of a little-known capability of the .Console driver, the partial character download feature. This is a "control" call to the driver (remember our previous sessions about console capabilities?), specifically control call 17. More information about how is works in in the Standard Device Drivers manual in the section on the Console driver. Basically, it allows you to change up to 8 character definitions "on the fly", and it does this very fast. To give you a feel for how this process works, let's try the following program:

```
10   DIM a%(511)
15   INVOKE"/basic/download.inv","/basic/request.inv"
20   q$=CHR$(34):array$="a%"
25   text40$=CHR$(16)+CHR$(1)
30   INPUT"Name of font file: ";flname$
35   name$=q$+flname$+q$
40   PERFORM getfont(@name$,@array$)
```

The lines above set up the array which will hold an alternate character set, invoke the necessary modules, and load the font into the "a%" array. Although

we don't need all these characters, it saves us the trouble of entering the definitions manually. The next section extracts two character definitions:

```
 45   ltr.a$="":ltr.b$=""
 50   FOR i=260 TO 263
 55     lt$=HEX$(a%(i))
 60     ltr.a$=ltr.a$+CHR$(TEN(MID$(lt$,1,2)))+CHR$(TEN(MID$(lt$,3,2)))
 65     NEXT i
 70   FOR i=264 TO 267
 75     lt$=HEX$(a%(i))
 80     ltr.b$=ltr.b$+CHR$(TEN(MID$(lt$,1,2)))+CHR$(TEN(MID$(lt$,3,2)))
 85     NEXT i
```

The routine above looks a little complicated, but it's really straight-forward. Since the SOS Control call mechanism in the "request.inv" invokable module uses a string variable as the parameter list, it is necessary to convert the contents of the integer array a% into equivalent ASCII characters. This is done in lines 55-60 and 75-80 by first converting the integer into HEX format, and then treating the result as a pair of two digit HEX numbers which are converted into ASCII characters by the CHR$ function. Notice also that since the integer format requires four locations for each character, the locations 260-263 and 264-267 correspond to upper-case A and B respectively (which probably explains the variable names).

Now that "ltr.a$" and "ltr.b$" contain the definitions for the two characters, we'll load the font into the character generator, turn on 40 column mode, clear the screen, and print several lines of A's and B's on the screen, like this:

```
  90   PERFORM loadfont(@array$)
  95   PRINT text40$:HOME
 100   FOR i=1 TO 10:PRINT"ABABABABABABABABABABAB":NEXT i
```

Next, we'll change those characters on the screen by using the partial character download that was mention before:

```
 110   name$=".console"
 120   ctrlist1$=CHR$(1)+CHR$(65)+ltr.b$
 130   ctrlist2$=CHR$(1)+CHR$(65)+ltr.a$
 150   PERFORM control(%17,@ctrlist1$)name$
 160   GET a$:IF ASC(a$)=27 THEN 500
 170   PERFORM control(%17,@ctrlist2$)name$
 180   GET a$:IF ASC(a$)<>27 THEN 150
```

In line 110, we define the name of the device driver to be called, and then lines 120 and 130 set up the parameter string for the control call. Notice that the format is the number of characters to be loaded, then the character number, followed by the character definition. If you load more than one character, the first value changes, and then each character definition in the string is preceeded

with its character number. In this case, character 65, which is normally an upper-case A, is being defined in "ctrlist1$" as a "B" and in "ctrlist2$" as an "A". Lines 150-180 do the character switching, with "GET" statements in-between to allow you to see what's going on. By slowly pressing any key, you can watch the A's turn to B's and back. Holding down a fast repeating key will give you an idea of just how fast these changes can take place. Pressing ESCape will allow the routine to clean up and end:

```
500   REM Restore Screen
510   PRINT CHR$(22);CHR$(14)
520   TEXT:HOME
530   nam$=q$+"/basic/standard"+q$
540   PERFORM getfont(@nam$,@array$):PERFORM loadfont(@array$)
550   PRINT CHR$(15);
560   END
```

Well, there you have it. The routine above could just as easily change every character on the screen to a different definition, since the change is in the character generator, not in the screen memory itself.

Some Relevance Rears its Ugly Head

What, you ask, does this have to do with "Bug-Mania" and character set animation? Good question, and one about to be answered by the next program. Remember, just because we changed one letter to another doesn't mean that's the only use of the principle. Last time we looked at redefining characters (we used control characters 21 through 26) to make little creatures to populate our game. The character definitions were created by the font editor from a few episodes back, or could be created with any font-editing program. The next program takes those character definitions and demonstrates how they can be used to make our little critters move. It is similar in structure to the previous program, with a more general purpose design, and has some similarities to the game program from last time:

```
 10   DIM a%(511)
 20   INVOKE"/basic/download.inv","/basic/request.inv"
 30   q$=CHR$(34):array$="a%"
 40   fg$=CHR$(19):bg$=CHR$(20)
 50   mblue$=CHR$(6):white$=CHR$(15)
 60   bw$=fg$+mblue$+bg$+white$
 70   text40$=CHR$(16)+CHR$(1)
 80   GOSUB 700:REM get font
 90   GOSUB 800:REM load up the bugs
100   GOSUB 600:REM set up screen
```

After the initialization in lines 10-70, three subroutines are called to set things up for the animation to follow. They look like this (in order of use):

```
700   RESTORE
705   head1$="":head2$="":body1$="":body2$="":tail1$="":tail2$=""
710   FOR i=1 TO 4
715     READ a%,b%,c%,d%,e%,f%
720     h1$=HEX$(a%):h2$=HEX$(b%):b1$=HEX$(c%):b2$=HEX$(d%)
725     t1$=HEX$(e%):t2$=HEX$(f%)
730     head1$=head1$+CHR$(TEN(MID$(h1$,1,2)))+CHR$(TEN(MID$(h1$,3,2)))
735     head2$=head2$+CHR$(TEN(MID$(h2$,1,2)))+CHR$(TEN(MID$(h2$,3,2)))
740     body1$=body1$+CHR$(TEN(MID$(b1$,1,2)))+CHR$(TEN(MID$(b1$,3,2)))
745     body2$=body2$+CHR$(TEN(MID$(b2$,1,2)))+CHR$(TEN(MID$(b2$,3,2)))
750     tail1$=tail1$+CHR$(TEN(MID$(t1$,1,2)))+CHR$(TEN(MID$(t1$,3,2)))
755     tail2$=tail2$+CHR$(TEN(MID$(t2$,1,2)))+CHR$(TEN(MID$(t2$,3,2)))
760     NEXT i
765   RETURN
780   DATA 7215,7215,14462,14462,0,0
785   DATA 32545,31555,27519,22399,1090,580
790   DATA 838,16156,32546,32546,16932,26640
795   DATA 15360,0,8806,4369,6144,0
```

The routine above uses some of the techniques from the last program defining the characters, except that this time we are reading the font definition from the data statements in lines 780-795. These numbers may look like gibberish, but, in the immortal words of many a programmer, "trust me". These values define our tiny creature's head, body and tail, and were, in fact, extracted from a font created by the font editor from Article 17. In the event you have a working version of that editor, or a similar one which produces system font files, you could use the "Bug-mania" font from Article 18, substituting the following lines for lines 700-795 above:

```
700   INPUT"Name of font file: ";flname$
702   name$=q$+flname$+q$
714   PERFORM getfont(@name$,@array$)
716   head1$="":head2$="":body1$="":body2$="":tail1$="":tail2$=""
718   FOR i=92 TO 95
720     hd$=HEX$(a%(i))
722     head1$=head1$+CHR$(TEN(MID$(hd$,1,2)))+CHR$(TEN(MID$(hd$,3,2)))
724     NEXT i
726   FOR i=104 TO 107
728     hd$=HEX$(a%(i))
730     head2$=head2$+CHR$(TEN(MID$(hd$,1,2)))+CHR$(TEN(MID$(hd$,3,2)))
732     NEXT i
734   FOR i=88 TO 91
736     bd$=HEX$(a%(i))
```

```
 738     body1$=body1$+CHR$(TEN(MID$(bd$,1,2)))+CHR$(TEN(MID$(bd$,3,2)))
 740     NEXT i
 742   FOR i=100 TO 103
 744     bd$=HEX$(a%(i))
 746     body2$=body2$+CHR$(TEN(MID$(bd$,1,2)))+CHR$(TEN(MID$(bd$,3,2)))
 748   FOR i=84 TO 87
 750     tl$=HEX$(a%(i))
 752     tail1$=tail1$+CHR$(TEN(MID$(tl$,1,2)))+CHR$(TEN(MID$(tl$,3,2)))
 754     NEXT i
 756   FOR i=96 TO 99
 758     tl$=HEX$(a%(i))
 760     tail2$=tail2$+CHR$(TEN(MID$(tl$,1,2)))+CHR$(TEN(MID$(tl$,3,2)))
 762     NEXT i
 764   RETURN
```

That seems like a lot of repeating, and it is, mostly for clarity. As set up, the lines above will extract the characters from last month's game set, if you have defined that font. By changing the parameters on the FOR-NEXT loops, and in the build routine below, you could use any set of characters from any font. To simplify the screen display, and to show how powerful the character download capability is, we'll build strings of creature characters to populate the screen:

```
 800   char$(0)="":char$(1)=CHR$(149):char$(2)=CHR$(150):
char$(3)=CHR$(151)
 810   m$=".23...123.1223..13.123.3.23...123.1223.."
 820   FOR i=1 TO 40:SUB$(m$,i,1)=char$(VAL(MID$(m$,i,1))):NEXT i
 830   RETURN
```

The routine above uses a technique borrowed from last article to create "m$" with the appropriate characters for the head, body and tail of the creatures. Notice that character values greater than 127 are used in order to map into the printable control character space. Characters 149-151 correspond to 21-23 in the standard ASCII set and are the same as used in last month's game.

Now on to the subroutine at 600, which sets up the screen and prints lots of bug-filled strings:

```
 600   PRINT text40$;bw$;:HOME
 610   PERFORM loadfont(@array$)
 620   FOR i=1 TO 11
 630     PRINT m$
 640     NEXT i
 650   RETURN
```

Now that the setup is done, it's on with the show:

```
 110   name$=".console"
```

```
120   ctrlist1$=CHR$(3)+CHR$(23)+head1$+CHR$(22)+body1$+CHR$(21)+tail1$
130   ctrlist2$=CHR$(3)+CHR$(23)+head2$+CHR$(22)+body2$+CHR$(21)+tail2$
140   ON KBD GOTO 200
150   PERFORM control(%17,@ctrlist1$)name$:FOR i=1 TO 10*pause:NEXT
160   PERFORM control(%17,@ctrlist2$)name$:FOR i=1 TO 10*pause:NEXT
170   GOTO 150
```

This section is also similar to the last program, except this time we load three characters at a time. Also, instead of requesting input between each character switch to slow the display changes down, FOR-NEXT loops are introduced, with a variable speed depending on the value of "pause". Exits and speed changes are taken by pressing keys which use the ON KBD routine in line 200:

```
200   OFF KBD
210   IF KBD=27 THEN 500
220   IF KBD>47 AND KBD<58 THEN pause= KBD-48
230   ON KBD GOTO 200
240   RETURN
```

Notice above that if ESCape (ASCII 27) is pressed, a jump is taken to line 500, where the program ends. If a number is pressed, then the pause factor is set to that numeric value. This allows you to see the speeds possible with this technique, while the program is running. Obviously, the routine could do lots of other interesting things. The cleanup routine, very similar to last time, looks like this:

```
500   PRINT CHR$(21);"="
510   PRINT CHR$(22);CHR$(14);
520   TEXT:HOME
530   nam$=q$+"/basic/standard"+q$
540   PERFORM getfont(@nam$,@array$):PERFORM loadfont(@array$)
550   PRINT CHR$(15);
560   END
```

The only noticeable change in the routine above is that the 'CHR$(21);"="' in line 500 sets all console options back to normal. While this is not necessary in this case, it is good practice, and will be required in some later programs.

That's All Fine, but was it Good for You?

Running the program above (the DATA statement version is recommended for starters) will demonstrate that this technique can produce some really fast in-place animation. With "pause" equal to zero, the critters will run in place so fast that you can hardly see their little legs move. However, running in place is not what we're after. The really interesting stuff is to use the character redefinition capability to produce smooth motion. Remember in the last article where we got animation by printing several different versions of the characters in different

places on the screen? That looked ok, but it suffered from one fact of life about printing characters, that is, there are only a limited number of horizontal locations where you can print them. In the case of the 40 column mode that we have been using, the characters appear to jerk from place to place, because there are only 40 places to draw them. But are we doomed to jerky animation? Not the MicroKids!

One obvious solution to the dilemma, and one which will be discussed in a moment, lies in using the hi-res graphics screen instead of the text screen that we have been using. Since you can draw any character at any dot location on the graphics screen, it's just a matter of repositioning and printing. Later on, and especially in the next article, we'll see how that can open up a whole new bag of tricks, but we're not finished with what text mode graphics can do for us yet.

It Ain't the Mode, it's the Motion

The text mode character sets are so fast that a whole new possibility exists to make motion out of multiple character definitions, and therein lies the solution to our dilemma. If we could create a definition of an object in all the phases of moving from one character cell to another, and could download these redefinitions rapidly, then the object would appear to move smoothly from one cell to another. Since a character cell is seven dots wide, this implies that to create smooth horizontal motion will require seven different definitions of the same character, each one moved over one row of pixels into the next character cell. All of which leads us to the following program:

```
10   DIM a%(511),ctrlist$(13),head1$(6),head2$(6)
20   INVOKE"/basic/download.inv","/basic/request.inv"
30   q$=CHR$(34):array$="a%"
40   fg$=CHR$(19):bg$=CHR$(20)
50   mblue$=CHR$(6):white$=CHR$(15)
60   bw$=fg$+mblue$+bg$+white$
70   text40$=CHR$(16)+CHR$(1)
80   GOSUB 700:REM get the head definitions
90   GOSUB 800:REM load the print line
100   GOSUB 600:REM set up screen
```

If the program above looks familiar, it should. With the exception of array dimension statements for the seven versions of the head, and the fourteen versions of the control list necessary to define the different versions of the head, it's the same as in the last program. As in the description of that program, let's take the subroutines in order:

```
700   RESTORE
710   FOR i=0 TO 6
715     head1$(i)="":head2$(i)=""
720     FOR j=0 TO 3
```

```
 725       READ a%:h$=HEX$(a%)
 730       head1$(i)=head1$(i)+CHR$(TEN(MID$(h$,1,2)))+CHR$(TEN(MID$
(h$,3,2)))
 735       NEXT j
 740     FOR j=0 TO 3
 745       READ a%:h$=HEX$(a%)
 750       head2$(i)=head2$(i)+CHR$(TEN(MID$(h$,1,2)))+CHR$(TEN(MID$
(h$,3,2)))
 755       NEXT j
 760     NEXT i
 765   RETURN
 770   DATA 7215,32545,838,15360,0,0,0,0
 772   DATA 14430,32322,1548,30720,0,256,1,0
 774   DATA 28732,31748,3096,28672,1,769,2,256
 776   DATA 24696,30728,6192,24576,258,1794,4,768
 778   DATA 16496,28688,12384,16384,773,3844,8,1792
 780   DATA 96,24608,24640,0,1803,7944,17,3840
 782   DATA 64,16448,16384,0,3607,16144,291,7680
```

The code in 700-765 above looks simpler (certainly shorter) than last program, but notice that there is lots more data. Simply put, each line of data defines two adjacent character cells. If you look back at the data in the previous program, you'll notice that the first four elements of line 770 in this program match the data in the first column in the other program. That's because 7215, 32545, 838, and 15360 are the four elements of the font definition which make up the creature's head. This is loaded by 720-735 into the first "head1$" definition. The next four numbers, all zeros in this case, make up the definition of the adjacent cell. This is logical, since the head starts in the first cell completely, with the second cell blank. The next row of data (line 772) describes what the head looks like when it is shifted one dot over in the first cell, and the second cell picks up the column of dots from the rightmost column of the first cell. The definition proceeds, each row representing the head shifted over one dot column into the adjacent cell. Since there is already a definition for the head completely in a cell, there is no need for an eighth definition, we can just switch the character definitions around. More on that later. Also, as you might have guessed, these numbers came from defining the characters with the font editor, and then printing out the results. Changing the program to accomodate font editor output directly would be done in a similar way to the alternate subroutine in the previous program.

Next, the subroutine at 800-830 builds the character string in a similar fashion to last program, with some notable exceptions:

```
 800   char$(0)=" ":char$(1)=CHR$(128):char$(2)=CHR$(129)
 810   m$="12121212121212121212121212121212121212121212"
 820   FOR i=1 TO 40:SUB$(m$,i,1)=char$(VAL(MID$(m$,i,1))):NEXT i
```

```
830   RETURN
```

Notice that this time we use ASCII 0 and 1 as our characters to redefine (converted to 128 and 129 to make them displayable). We don't need seven different characters, that will be handled by successively redefining the two on the screen. Note also that we fill up "m$" completely, because we want smooth motion from one side of the screen to the other.

Next comes the routine to put our character strings on the screen. This time we've chosen to fill nearly the whole screen up with characters, to wit:

```
600   PRINT text40$;bw$
610   HOME:PRINT:PRINT
620   FOR i=1 TO 21
630     PRINT m$;
640     NEXT i
650   RETURN
```

Ok, now to make all these little heads pay attention:

```
110   name$=".console"
115   FOR i=0 TO 6
120     ctrlist$(i)=CHR$(2)+CHR$(0)+head1$(i)+CHR$(1)+head2$(i)
125     NEXT i
130   FOR i=7 TO 13
135     ctrlist$(i)=CHR$(2)+CHR$(0)+head2$(i-7)+CHR$(1)+head1$(i-7)
140     NEXT i
145   ON KBD GOTO 200
150   FOR i=0 TO 13:PERFORM control(%17,@ctrlist$(i))name$:NEXT
155   GOTO 150
```

This time we create a series of control strings for efficiency. The first seven definitions in "ctrlist$" describe how the head moves from character 0 to character 1. The second seven (in lines 130-140) show how the head looks moving from 1 to 0. Recall that since the screen lineup is 010101... that steady cycling through the definitions will create the desired motion. Line 150 and 155 accomplish this, endlessly repeating the whole sequence.

The ON KBD routine is simpler this time, and just gives us a way out:

```
200   OFF KBD
210   IF KBD=27 THEN 500
220   ON KBD GOTO 200
230   RETURN
```

The cleanup routine at 500 is identical to the one in the last program. Just copy it and you'll exit to the normal world properly with the ESC character.

At Long Last, Bug

When you run this, the effect is somewhat amazing. With a simple Basic program, you're creating smooth animation which takes quite a bit of assembly language on an Apple][, IBM PC, etc. In fact, even the best assembly programmers would be hard-pressed to create smooth motion over such a large area of the screen. But wait, you say. You've seen smooth animation on the Apple III before. What about that running horse demo on the System Demo disk? If you haven't seen it, take a moment to run the demo that comes with each Apple III and select the "horse" section. You will see magnificent full color animation, that you probably always thought was done on the high-res screen. Think again, bucko! That demo uses the save character set redefinition techniques that we've just been discussing. If you are patient enough, you might create the horse by defining all the characters, each a small piece of the big puzzle. By creating multiple definitions of these characters, both in place and in motion, you can repeat that display yourself, using the principles in this article.

RAMbling Onward

As you might guess, the paragraph above contains enough suggestions to last a lifetime, but since it's only one short month until next month's article, we had best move on to the promised treatment of the high-resolution graphics mode. As was mentioned previously, smooth motion on the high-res screen is trivial, and there's nothing like a trivial program to demonstrate that fact:

```
10   INVOKE"/basic/bgraf.inv"
15   OPEN#1,".grafix"
20   m$=" **  ****  **  ****  *** * ****   ** ***"
25   PERFORM initgrafix
30   PERFORM grafixmode(%3,%1)
35   PERFORM fillcolor(%4):PERFORM pencolor(%13)
40   PERFORM fillport
45   PERFORM grafixon
50   FOR i=0 TO 139
55     PERFORM moveto(%i,%180):PRINT#1;m$
60     NEXT i
65   GET a$:IF ASC(a$)<>27 THEN 50
100   PERFORM release:PERFORM release:PERFORM release
105   CLOSE:INVOKE
110   TEXT
115   END
```

This just prints the string "m$" starting at each successive location on the high-res screen, in this case the 140X192 16 color mode. One or two things should be noted here, which will become more important later, especially in the next article. First, notice that we can print off the screen without any apparent problem, and without wrapping onto the next line. Second, notice that the display seems to move faster as less and less of the string is actually printed in the display area. Watch this closely, its a clue to what's happening in the graphics driver. Another thing is interesting and it's worth experimenting with. Try changing the text character set before opening the ".grafix" driver, using the techniques previously described. This allows you to print special characters like the bug's head onto the graphics screen, just as you would any other character.

If positioning was the only virtue of the graphics driver, then it would be not nearly as useful as we would want. The next little program will give a clue to some of the graphic's drivers special capabilities, which will be used fully in the next exciting episode of this long-winded serial. Try the following:

```
10   INVOKE"/basic/bgraf.inv"
15   OPEN#1,".grafix"
20   m$=" **  ****  **  ****  *** * ****   ** *****"
25   PERFORM initgrafix
30   PERFORM grafixmode(%3,%1)
35   PERFORM pencolor(%13)
40   PERFORM fillcolor(%4):PERFORM fillport
45   PERFORM setctab(%4,%9,%9):PERFORM setctab(%13,%9,%9)
50   PERFORM fillcolor(%9)
55   PERFORM viewport(%0,%6,%0,%191):PERFORM fillport
60   PERFORM viewport(%133,%139,%0,%191):PERFORM fillport
65   PERFORM fillcolor(%4)
70   PERFORM viewport(%0,%139,%0,%191)
75   PERFORM grafixon
80   FOR j=7 TO-77 STEP-1
85     PERFORM moveto(%j,%180):PRINT#1;m$
90     NEXT j
95   GET a$:IF ASC(a$)<>27 THEN 80
100   PERFORM release:PERFORM release:PERFORM release
105   CLOSE:INVOKE
110   TEXT
115   END
```

Getting Your (Color) Priorities Straight

Looks pretty much like the last one, right? Not right. Notice that our primary colors are 4 and 13, the fillcolor and pencolor respectively. Notice also the color priority definition in line 45. This says that anytime color 4 or color 13 are

printed over color 9 (which is the color of two border strips that are set up at each edge of the screen) that the result should remain color 9. Run the program and see how this works. You should also notice that this time the print string runs backwards, not forwards. The color priority table is a very powerful feature of the Apple III graphics driver, one which is exploited very little by programmers. Next time we will work this feature to death, and in the process do things which would challenge the best of the high-res jocks on less capable machines. Until then, remember that a pixel a day keeps boredom away, which gives you 107,520 days to enjoy your Apple III!

```
 50   REM 7, 11 and 14 are best background colors
 100   OPEN#1,".grafix"
 110   INVOKE"/basic/bgraf.inv"
 120
black%=0:blue%=6:orange%=9:green%=12:white%=15:dgreen%=4:brown%=8:grey%=
10:yellow%=13
 130   PERFORM grafixmode(%3,%1)
 140   PERFORM initgrafix
 150   vector(1)=dgreen%:vector(2)=brown%:vector(3)=grey%:
vector(4)=yellow%
 160   PERFORM setctab(%dgreen%,%blue%,%blue%)
 170   PERFORM setctab(%dgreen%,%orange%,%orange%)
 180   PERFORM setctab(%dgreen%,%green%,%green%)
 190   PERFORM setctab(%dgreen%,%white%,%white%)
 200   PERFORM setctab(%brown%,%orange%,%orange%)
 210   PERFORM setctab(%brown%,%green%,%green%)
 220   PERFORM setctab(%brown%,%white%,%white%)
 230   PERFORM setctab(%grey%,%green%,%green%)
 240   PERFORM setctab(%grey%,%white%,%white%)
 250   PERFORM setctab(%yellow%,%white%,%white%)
 255   INPUT"Background color number: ";a$
 260   a=VAL(a$):IF a$="" OR a<0 OR a>15 THEN 510
 265   PERFORM grafixon
 270   PERFORM fillcolor(%a):PERFORM fillport
 300   PERFORM viewport(%20,%30,%40,%170):PERFORM
fillcolor(%black%):PERFORM fillport
 310   PERFORM viewport(%35,%40,%30,%160):PERFORM
fillcolor(%blue%):PERFORM fillport
 320   PERFORM viewport(%52,%65,%40,%170):PERFORM
fillcolor(%orange%):PERFORM fillport
 330   PERFORM viewport(%77,%92,%35,%175):PERFORM
fillcolor(%green%):PERFORM fillport
 340   PERFORM viewport(%110,%120,%20,%170):PERFORM
fillcolor(%white%):PERFORM fillport
 400   PERFORM viewport(%0,%139,%0,%192)
```

```
405   FOR i=1 TO 4
410     FOR j=1 TO 10
415       horiz=(i-1)*45+j*4
420       PERFORM moveto(%140,%horiz)
430       PERFORM pencolor(%vector(i))
440       PERFORM lineto(%0,%96)
450       NEXT j
460     NEXT i
500   GET a$:IF ASC(a$)<>27 THEN TEXT:GOTO 255
510   TEXT
520   INVOKE:CLOSE
530   PERFORM release:PERFORM release:PERFORM release
540   END
1160   PERFORM setctab(%dgreen%,%blue%,%blue%)
1170   PERFORM setctab(%dgreen%,%orange%,%orange%)
1180   PERFORM setctab(%dgreen%,%green%,%green%)
1190   PERFORM setctab(%dgreen%,%white%,%white%)
1200   PERFORM setctab(%brown%,%orange%,%orange%)
1210   PERFORM setctab(%brown%,%green%,%green%)
1220   PERFORM setctab(%brown%,%white%,%white%)
1230   PERFORM setctab(%grey%,%green%,%green%)
1240   PERFORM setctab(%grey%,%white%,%white%)
1250   PERFORM setctab(%yellow%,%white%,%white%)
1260   RETURN
```

```
5   DIM shape%(1,15),x%(127),y%(127)
10   INVOKE"/basic/bgraf.inv"
15   OPEN#1,".grafix"
20   GOSUB 1000
25   PERFORM initgrafix
30   PERFORM grafixmode(%3,%1)
35   PERFORM fillcolor(%4):PERFORM pencolor(%13)
40   PERFORM fillport
45   PERFORM viewport(%0,%139,%0,%191)
50   PERFORM grafixon
55   f%=4:s%=16:z%=0
60   x%(8)=-3:x%(21)=3:y%(11)=3:y%(10)=-3
65   i=70:j=90
70   PERFORM moveto(%i,%j)
75   IF r%=0 THEN r%=16:ELSE r%=0
80   PERFORM drawimage(@shape%(0,0),%f%,%r%,%z%,%s%,%s%)
85   GET a$:a=ASC(a$)
```

```
90   IF a<>27 THEN i=i+x%(a):j=j+y%(a):GOTO 70
100   PERFORM release:PERFORM release:PERFORM release
105   CLOSE:INVOKE
110   TEXT
115   END
1000   FOR j=0 TO 15:FOR i=0 TO 1:READ shape%(i,j):NEXT:NEXT
1010   RETURN
2000   DATA 0,0
2005   DATA 0,0
2010   DATA 0,0
2015   DATA 0,1984
2020   DATA 1984,2080
2025   DATA 2080,4752
2030   DATA 4752,4112
2035   DATA 4112,5008
2040   DATA 5008,2080
2045   DATA 2080,1984
2050   DATA 1984,2336
2055   DATA 2976,2720
2060   DATA 1344,4752
2065   DATA 0,0
2070   DATA 0,0
2075   DATA 0,0
```

```
10   DIM mshape%(1,15),zshape%(1,31)
15   INVOKE"/basic/bgraf.inv"
20   OPEN#1,".grafix"
25   GOSUB 1000
35   PERFORM initgrafix
40   PERFORM grafixmode(%3,%1)
45   PERFORM pencolor(%4)
50   PERFORM fillcolor(%7):PERFORM fillport
55   PERFORM viewport(%40,%100,%15,%130)
60   PERFORM fillcolor(%13):PERFORM fillport
65   PERFORM grafixon
100   s%=16:t%=32:z%=0:f%=4
105   FOR k=0 TO 100
110     j=COS(k/5+2)*53+88
115     l=SIN(k/10)*30+58
120     m=SIN(k/10)*70+85
125     r%=s%*(r%=z%)
130     PERFORM moveto(%k+28,%j)
135     PERFORM drawimage(@zshape%(0,0),%f%,%r%,%z%,%s%,%t%)
```

```
140     PERFORM moveto(%k+14,%l)
145     PERFORM drawimage(@mshape%(0,0),%f%,%r%,%z%,%s%,%s%)
150     PERFORM moveto(%k,%m)
155     PERFORM drawimage(@zshape%(0,0),%f%,%r%,%z%,%s%,%t%)
160     NEXT k
200   GET a$:IF ASC(a$)<>27 THEN 105
300   PERFORM release:PERFORM release:PERFORM release
310   CLOSE:INVOKE
320   TEXT
330   END
1000   FOR j=0 TO 15:FOR i=0 TO 1:READ mshape%(i,j):NEXT:NEXT
1010   FOR j=0 TO 15:FOR i=0 TO 1:zshape%(i,j+8)=mshape%(i,j):NEXT:NEXT
1020   RETURN
2000   DATA 0,0
2005   DATA 0,0
2010   DATA 0,0
2015   DATA 0,1984
2020   DATA 1984,2080
2025   DATA 2080,4752
2030   DATA 4752,4112
2035   DATA 4112,5008
2040   DATA 5008,2080
2045   DATA 2080,1984
2050   DATA 1984,2336
2055   DATA 2976,2720
2060   DATA 1344,4752
2065   DATA 0,0
2070   DATA 0,0
2075   DATA 0,0
```

```
10   DIM shape%(7,15),tshape%(7,31)
15   INVOKE"/basic/bgraf.inv","/basic/request.inv"
20   OPEN#1,".grafix"
25   file$="next.shape":GOSUB 1000
30   FOR j=0 TO 15:FOR i=0 TO 7:tshape%(i,j+8)=shape%(i,j):NEXT:NEXT
35   PERFORM initgrafix
40   PERFORM grafixmode(%3,%1)
45   PERFORM pencolor(%4)
50   PERFORM fillcolor(%7):PERFORM fillport
55   PERFORM viewport(%40,%100,%15,%130)
60   PERFORM fillcolor(%13):PERFORM fillport
65   PERFORM grafixon
```

```
100   s%=16:t%=32:z%=0
105   FOR k=0 TO 100
110     j=COS(k/5+2)*53+88
115     l=SIN(k/10)*30+58
120     m=SIN(k/10)*70+85
125     r%=t%+s%*(r%=t%)
130     PERFORM moveto(%k+28,%j)
135     PERFORM drawimage(@tshape%(0,0),%s%,%r%,%z%,%s%,%t%)
140     PERFORM moveto(%k+14,%l)
145     PERFORM drawimage(@shape%(0,0),%s%,%r%,%z%,%s%,%s%)
150     PERFORM moveto(%k,%m)
155     PERFORM drawimage(@tshape%(0,0),%s%,%r%,%z%,%s%,%t%)
160     NEXT k
200   GET a$:IF ASC(a$)<>27 THEN 105
300   PERFORM release:PERFORM release:PERFORM release
310   CLOSE:INVOKE
320   TEXT
330   END
1000   ON ERR GOTO 1030
1010   array$="shape%":OPEN#3,file$
1020   IF TYP(3)=1 THEN READ#3;filtyp,ch,cw,sl:IF filtyp=1 THEN 1070
1030   OFF ERR:CLOSE#3
1040   IF TYP(3)=0 THEN DELETE file$
1050   error=1:REM Not a shape file
1060   RETURN
1070   READ#3,1:PERFORM filread(%3,@array$,%256,@ret%):OFF ERR:CLOSE#3
1080   IF ret%<>256 THEN error=2:RETURN:REM Shape definition is invalid
1090   error=0:RETURN:REM Shape loaded
```

Exploring Business Basic, Part XX

Greetings, BASIC buffs! This month's article continues the never-ending (it seems that way) saga of graphics capabilities on the Apple III. It's only fair to point out (as several of you have already) that many of the techniques shown in this series are useful from any language on the III. Pascal in particular can make SOS calls and communicate to drivers through its own set of proceedures, similar to the Business Basic invokables. In fact, the Pascal equivalent of BGRAF is called PGRAF (clever, those Apple programmers!). Since the ideas in this series utilize the unique hardware and operating system capabilities of the III, rather than just those of Business Basic, you can adapt most of the material whether you are working in Pascal, COBOL, FORTRAN or Assembler.

That's about it for "News and Views" this month. We've got a lot of material to go over, and after several months at this, it will be assumed that you are pretty familiar with the concepts. This issue will cover one last technique (oh, no! not the Last Technique!) for character graphic animation, and then plunge headlong into the Apple III bit-mapped display, most notably the 140X192 sixteen color mode.

A Last Issue from Last Time

As you may remember, one of the major events from last time was the use of the "character download" capability of the .console driver to create smooth animation by rapid switching of character definitions. We proved how powerful the technique was by smoothly scrolling hundreds of little bug heads from one side of the screen to the other. The question that somewhat naturally came up was "How can I move just one head across the screen?". Curiously enough, its a little harder, enough so that its worth showing a program to accomplish it. If you read last month's article, you'll notice a fair amount of similarity in structure with other programs in that treatise.

The trick last time to moving lots of bug heads smoothly was to print alternating sets of character 128 and character 129 across the screen and then download definitions of each character which showed the head making a step-by-step transition from one character cell to the other. Since a character cell is seven dots wide, it takes eight steps to go from completely in one cell to completely in the next one. By reversing the process for every other cell, the illusion of smooth motion is created. To create the character cell definitions, we used the font and shape editor which was described in the February issue (part 17). However, in case you missed that issue, or couldn't summon the strength to type the whole thing in, the following program uses data statements to define the bug head transitions. Feel free to edit your own head definitions and substitute the approriate routine if you want.

As was mentioned above, moving a single head across the screen is a little trickier than flipping one character definition to the next. The simplest solution is to print a string of characters across the screen, each one a different character number, but initially all defined to appear as a blank. The program can then redefine the characters, one at a time, and give the appearance of motion across the screen, as each successive character is given the head definition. The following program illustrates one way to use this approach to solve the problem:

```
 10   DIM a%(511),ctrlist$(7),head1$(6),head2$(6)
 20   INVOKE"/basic/download.inv","/basic/request.inv"
 30   q$=CHR$(34):array$="a%":name$=".console"
 40   fg$=CHR$(19):bg$=CHR$(20)
 50   mblue$=CHR$(6):white$=CHR$(15)
 60   bw$=fg$+mblue$+bg$+white$
 70   text40$=CHR$(16)+CHR$(1)
```

Line 10 defines various arrays to be used in the program. 'Ctrlist$' contains the character redefinitions for downloading to the character generator, and the two 'head$' arrays contain definitions of the bug head, coming and going. The other lines above establish constants which will be used later.

Now, to initialize the data and get ready to scroll, we will use several routines, which will be discussed in turn:

```
  80   GOSUB 700:REM get the head definitions
  90   GOSUB 800:REM load the print line
 100   GOSUB 900:REM set up control strings
 110   GOSUB 600:REM set up screen
```

The first routine loads the head definitions, in this case from DATA statements:

```
 700   RESTORE
 710   FOR i=0 TO 6
 715     head1$(i)="":head2$(i)=""
 720     FOR j=0 TO 3
 725       READ a%:h$=HEX$(a%)
 730       head1$(i)=head1$(i)+CHR$(TEN(MID$(h$,1,2)))+CHR$(TEN(MID$
(h$,3,2)))
 735       NEXT j
 740     FOR j=0 TO 3
 745       READ a%:h$=HEX$(a%)
 750       head2$(i)=head2$(i)+CHR$(TEN(MID$(h$,1,2)))+CHR$(TEN(MID$
(h$,3,2)))
 755       NEXT j
 760     NEXT i
 765   RETURN
```

```
770   DATA  7215,32545,  838,15360,    0,    0,    0,    0
772   DATA 14430,32322, 1548,30720,    0,  256,    1,    0
774   DATA 28732,31748, 3096,28672,    1,  769,    2,  256
776   DATA 24696,30728, 6192,24576,  258, 1794,    4,  768
778   DATA 16496,28688,12384,16384,  773, 3844,    8, 1792
780   DATA    96,24608,24640,    0, 1803, 7944,   17, 3840
782   DATA    64,16448,16384,    0, 3607,16144,  291, 7680
```

The routine above is identical to one discussed last time, so you're on your own. If you want to use a font editor to create the heads, just load up the 'head$' arrays in the same sequence as found in the DATA statements (which are arranged as they would appear in an actual font definition).

Next comes the routine to create a print string for the screen, and to redefine all the characters in it to blanks:

```
800   line$=""
810   FOR i=0 TO 31
820     line$=line$+CHR$(128+i)
830     NEXT i
840   cr0$=CHR$(0)
850   ctrl$=CHR$(1)+cr0$+cr0$+cr0$+cr0$+cr0$+cr0$+cr0$+cr0$+cr0$
860   FOR i=0 TO 31
870     SUB$(ctrl$,2,1)=CHR$(i)
880     PERFORM control(%17,@ctrl$)name$
890     NEXT i
895   RETURN
```

Notice in lines 810-830 that line$ is built containing all the characters from 128 to 159 which are the displayable versions of character definitions 0 to 31. We could have used more characters, but it is necessary to skip character number 32 (space) or risk redefining every space character on the screen. As long as we just redefine control characters (0-31), we won't disturb anything important. Of course, with some thought, you can build 'line$' to any reasonable length.

Next comes the routine which defines the character download strings, defining the successive versions of the characters:

```
900   FOR i=1 TO 6
905     ctrlist$(i)=CHR$(2)+CHR$(0)+head1$(i)+CHR$(1)+head2$(i)
910     ctrb$(i)=CHR$(1)+CHR$(0)+head2$(i)
915     ctre$(i)=CHR$(1)+CHR$(31)+head1$(i)
920     NEXT i
925   ctrlist$(7)=CHR$(2)+CHR$(0)+head2$(0)+CHR$(1)+head1$(0)
930   ctrb$(7)=CHR$(1)+CHR$(0)+head1$(0)
935   ctre$(7)=CHR$(1)+CHR$(31)+head2$(0)
940   RETURN
```

Note that in addition to the 'ctrlist$' definitions above, which define the transitions of the characters in the middle of the screen, there are two special arrays, 'ctrb$' and 'ctre'. 'Ctrb$' is the head definition as it appears onto the screen in the leftmost character position, while 'ctre$' is the definition of the rightmost character, as it disappears.

Lastly comes the screen setup routine:

```
 600   PRINT text40$;bw$
 605   PRINT CHR$(21);"9";
 610   HOME:PRINT:PRINT
 620   RETURN
```

This just turns on the forty column mode, sets the color to blue on a white background, turns off character wrap (line 605), and clears the screen to white.

Now the fun begins:

```
 120   PRINT line$;
 150   ON KBD GOTO 200
 160   FOR i=1 TO 7:PERFORM control(%17,@ctrb$(i))name$:NEXT
 165   FOR x=0 TO 30:FOR i=1 TO 7
 170       SUB$(ctrlist$(i),2,1)=CHR$(x):SUB$(ctrlist$(i),11,1)=CHR$
(x+1)
 175       PERFORM control(%17,@ctrlist$(i))name$
 180       NEXT:NEXT
 185   FOR i=1 TO 7:PERFORM control(%17,@ctre$(i))name$:NEXT
 190   GOTO 160
```

We stated earlier that the animation would occur by successively redefining our string of characters. Line 120 prints the character string on the screen, and line 150 sets up an ON KBD jump to exit from the scrolling. Scrolling begins in line 160 by successively redefining the first character in the string through the seven phases required to bring the head fully into the first character position. Line 165-180 then set up a major loop to proceed through the characters in 'line$' (remember that they are now printed on the screen), taking each through the seven phase redefinition necessary for the smooth scrolling. Notice that the same 'ctrlist$' definition is used each time, with the appropriate character numbers plugged in line 170. If you are unsure as to how the 'PERFORM control' statement works, review last month's article, the section in the Standard Device Drivers Manual on .Console control functions, and the 'Request.inv' documentation on the Basic disk.

All that remains now of our program is the keyboard service routine at line 200 and the cleanup and exit routine at line 500, to wit:

```
 200   OFF KBD
 210   IF KBD=27 THEN 500
 220   ON KBD GOTO 200
```

```
 230   RETURN

 500   PRINT CHR$(21);"="
 510   PRINT CHR$(22);CHR$(14);
 520   TEXT:HOME
 530   nam$=q$+"/basic/standard"+q$
 540   PERFORM getfont(@nam$,@array$):PERFORM loadfont(@array$)
 550   PRINT CHR$(15);
 560   END
```

Well, that's it. When you run this program, the little creature's head should appear on the left side of the screen, move smoothly to the right, and disappear somewhere around the 30th character position, only to reappear again on the left of the screen. One thing is noticeable, however. If you watch closely, the head appears to zip onto and off the screen much faster than it chugs across the main part of the screen. This is because the routines at line 160 and line 185 are much faster than the main routine in lines 165-180. Although the routine can be speeded up somewhat, more drastic measures are required to make it substantially faster. The tradeoff, as usual, is memory space for tables. By pre-storing the results of the various string substitutions in a large string array, the whole sequence can be rewritten as a simple loop. We will declare a string array 'zoom$' to accomplish this, with the following changes to the program above:

```
 10   DIM a%(511),ctrlist$(7),head1$(6),head2$(6),zoom$(231)

 160   FOR i=1 TO 231:PERFORM control(%17,@zoom$(i))name$:NEXT
 165   GOTO 160

 940   FOR z=1 TO 7:zoom$(z)=ctrb$(z):NEXT
 945   FOR x=0 TO 30:FOR i=1 TO 7
 950       SUB$(ctrlist$(i),2,1)=CHR$(x):SUB$(ctrlist$(i),11,1)=CHR$
(x+1)
 955       zoom$(7*(x+1)+i)=ctrlist$(i)
 960       NEXT:NEXT
 965   FOR z=225 TO 231:zoom$(z)=ctre$(z-224):NEXT
 970   RETURN
```

Notice that we have added code in lines 940-970 to put the various string values into the 'zoom$' array, and therefore line 160 now performs the entire sequence, much faster than before (you can now delete lines 170-190). The difference in speed should be really noticeable.

The uses of this basic idea are practically unlimited. For example, by splitting up the string and printing it in various places on the screen, you can cause the head to move around, disappear from one spot, only to re-appear somewhere else. Have fun!

Can't Tell One Pixel from Another Without a Bit Map

While the preceding retrospective into character graphics was important to clear up some issues, the real purpose of this article was to get heavily involved in the graphics modes of the Apple III, called a "bit-mapped" display, since the image is created by reading out bits of data from certain areas of memory and translating them to dots (pixels) on the display screen. As a reminder, let's look at the program we concluded with last time, which gives a hint as to an important capability of the Apple III graphics driver, the setting of priorities in the 'color table'. Remember that all communication to and from the actual bit-map is via the .grafix driver, and the Bgraf.inv routine is used to make the functions of the driver easier to perform. The Bgraf procedure 'setctab' is used to change single color entries in the table in a much easier manner than the .grafix driver permits directly. Type in the following program, and we'll begin our exploration:

```
 10   INVOKE"/basic/bgraf.inv"
 15   OPEN#1,".grafix"
 20   m$=" **  ****  **  ****  *** * ****   ** *****"
 25   PERFORM initgrafix
 30   PERFORM grafixmode(%3,%1)
 35   PERFORM pencolor(%13)
 40   PERFORM fillcolor(%4):PERFORM fillport
 45   PERFORM setctab(%4,%9,%9):PERFORM setctab(%13,%9,%9)
 50   PERFORM fillcolor(%9)
 55   PERFORM viewport(%0,%6,%0,%191):PERFORM fillport
 60   PERFORM viewport(%133,%139,%0,%191):PERFORM fillport
 65   PERFORM fillcolor(%4)
 70   PERFORM viewport(%0,%139,%0,%191)
 75   PERFORM grafixon
 80   FOR j=7 TO-77 STEP-1
 85     PERFORM moveto(%j,%180):PRINT#1;m$
 90     NEXT j
 95   GET a$:IF ASC(a$)<>27 THEN 80
100    PERFORM release:PERFORM release:PERFORM release
105    CLOSE:INVOKE
110    TEXT
115    END
```

Setting your Priorities

In the little demonstration of color priorities above, we set the primary colors to be 4 and 13, the fillcolor and pencolor respectively, in lines 35 and 40, and fill

the screen with color 4 (dark green) . The color table definition in line 45 says that anytime color 4 or color 13 are printed over color 9, the result will remain color 9. Lines 50-65 use the window capability to set up two orange (color 9) borders. Without the changes to the default color table, anything printed in the first or last column of the screen would destroy the borders. A fairly complete description of this process can be found in the Device Drivers manual and in the Business Basic manual (Volume 2), but running the program is the best way to see how this works. The loop in lines 80-90 makes the print string run backwards, one pixel position at a time. As the string runs off the left side of the screen, the priority established in the color table insures that neither the fill or pen color will disturb the color 9 bars at the edge of the screen.

This very powerful feature of the Apple III graphics driver is exploited very little by programmers, but will be the subject of much of the rest of this article. One comment is important before we proceed, however. If you don't have a color monitor, there is no need for dispair. The Apple III automatically translates color output into sixteen shades of gray (or green) on your monochrome display, and the color values in the following examples were arrived at to make sure that they would look OK without color. So, get a good grip on your 'BGRAF.INV' module, 'cause here we go!

Becoming a Fan of Hi-res Graphics

The program below illustrates the abilities of the color priority table on a much grander scale, and even suggests how the capability can be used to give the illusion of depth to an image on the screen. As explained in the Basic manual section on 'SETCTAB', every pixel to be plotted on the screen is first passed through the color table, and converted, if necessary, before being actually drawn. The word "priority" is somewhat of a misnomer, actually, since the entrys in the table can produce any color as a result of plotting one color over another. Its just as easy to specify that plotting dark green over orange will produce white as to say that orange will be unaffected. In addition, the default is what you would expect, that is, the color you plot with is the color you get on the screen.

Without further ado, let's look at the program:

```
  50   REM 7, 11 and 14 are best background colors
 100   OPEN#1,".grafix"
 110   INVOKE"/basic/bgraf.inv"
 120   black%=0:blue%=6:orange%=9:green%=12:white%=15
 130   dgreen%=4:brown%=8:grey%=10:yellow%=13
 140   vector(1)=dgreen%:vector(2)=brown%:vector(3)=grey%:
vector(4)=yellow%
 150   PERFORM grafixmode(%3,%1)
 160   PERFORM initgrafix
```

```
170   GOSUB 1000
```

The program opens with the usual initialisation, and defines a number of variables as color constants in lines 120-140. Line 150 sets the graphics mode to three, the 140X192 color screen. This is the unrestricted sixteen color mode, which will be used for the rest of this article. After initializing the graphics screen, we set up the changes to the color table in a routine at line 1000:

```
1000   PERFORM setctab(%dgreen%,%blue%,%blue%)
1010   PERFORM setctab(%dgreen%,%orange%,%orange%)
1020   PERFORM setctab(%dgreen%,%green%,%green%)
1030   PERFORM setctab(%dgreen%,%white%,%white%)
1040   PERFORM setctab(%brown%,%orange%,%orange%)
1050   PERFORM setctab(%brown%,%green%,%green%)
1060   PERFORM setctab(%brown%,%white%,%white%)
1070   PERFORM setctab(%grey%,%green%,%green%)
1080   PERFORM setctab(%grey%,%white%,%white%)
1090   PERFORM setctab(%yellow%,%white%,%white%)
1100   RETURN
```

In the case above, we are using the color table to establish priorities for the 'vector' colors to be drawn. As you can see, dark green won't affect anything but the background, since drawing dark green over blue, orange, green or white will have no effect. Brown, however, will draw over blue, but loses to orange, green and white. Grey draws over blue and orange, but has no effect on green and white. Yellow effects everything but white. These statements are easy to understand, and could be programmed in simple background situations, but imagine what it would be like to draw lines or shapes on complex backgrounds! Checking on every spot on the screen to see what color is already there would be incredibly time-consuming. Since the color table is built into the .grafix driver, using it causes the check to be done at assembly language speeds, without having to worry about it.

Oh, well, enough praise for .grafix. To continue:

```
200   INPUT"Background color number: ";a$
210   a=VAL(a$):IF a$="" OR a<0 OR a>15 THEN 510
```

To further see the effect of the color table, the program allows you to set the overall background color for the screen. You should choose this carefully, since the color table will affect the results of certain choices. Colors 7, 11 and 14 should give good results.

Next, we'll use the viewport, fillport combinations to create various color bars on the screen, after first turning on the screen and clearing to the background color you have just chosen:

```
220   PERFORM grafixon
230   PERFORM fillcolor(%a):PERFORM fillport
```

```
300   PERFORM viewport(%20,%30,%40,%170)
310   PERFORM fillcolor(%black%):PERFORM fillport
320   PERFORM viewport(%35,%40,%30,%160)
330   PERFORM fillcolor(%blue%):PERFORM fillport
340   PERFORM viewport(%52,%65,%40,%170)
350   PERFORM fillcolor(%orange%):PERFORM fillport
360   PERFORM viewport(%77,%92,%35,%175)
370   PERFORM fillcolor(%green%):PERFORM fillport
380   PERFORM viewport(%110,%120,%20,%170)
390   PERFORM fillcolor(%white%):PERFORM fillport
```

Note that normally you would set up the screen, and then turn on the display (i.e. move statement 220 to after 390). In this case, its worth looking at how the color bars are set up, especially if you use different background colors than those suggested. The lines above could be replaced by some data statements and a loop for more compactness, since each set of two statements is identical except for parameters, but this way you get a feel for exactly what's going on. Next, we draw vectors over our landscape, and observe the color table effects:

```
400   PERFORM viewport(%0,%139,%0,%192)
410   FOR i=1 TO 4
420     FOR j=1 TO 10
430       horiz=(i-1)*45+j*4
440       PERFORM moveto(%140,%horiz)
450       PERFORM pencolor(%vector(i))
460       PERFORM lineto(%0,%96)
470       NEXT j
480     NEXT i
500   GET a$:IF ASC(a$)<>27 THEN TEXT:GOTO 255
```

The routine above sets the viewport to the whole screen, and then uses a double loop to draw lines from the point x=0,y=96 (middle of the left side of the screen) to various points on the right hand side. The effect is somewhat like a fan, or rays projected from a single source. Because of the color priorities established, the effect is quite dramatic, since the rays appear to go behind some objects, and in front of others. Pressing Escape in line 500 terminates the program, like so:

```
510   TEXT
520   PERFORM release:PERFORM release:PERFORM release
530   INVOKE:CLOSE
540   END
```

Running the program above with various background colors and various settings of the color table will allow you to experiment with the SETCTAB proceedure enough to get to know its capabilities. You might try setting

different result colors, for example, setting some to the background color, and observing the effects.

The Bugs are Back

Although the program listed above is interesting, even dramatic in its own way, you're paying to see the creatures from space, right? So let's bring on the bugs! Actually, it is useful to look at a combining of the techniques which we have already discussed with the new capabilities of the bit-mapped graphics display. The following program introduces a new creature, somewhat larger than his character graphics ancestors, which was originally created using the Shape Editor from the February issue. We will use this new kind of "bug" (which really resembles a squid), to begin our discussion of animation on the hi-res screen. The program below defines the creature, and lets us move him (ugly creatures are always male) around on the screen. We begin with the usual declarations:

```
  5   DIM mshape%(1,15),x%(255),y%(255)
 10   INVOKE"/basic/bgraf.inv"
 15   OPEN#1,".grafix"
 20   GOSUB 1000
```

Mshape% in line 5 is the array which will contain the creature's shape definition, in two parts. The first part (column) will show the tenacles extended, the second column defines the tenacles retracted. This allows simple animation, along with movement. The x% and y% arrays will be covered in a minute. After invoking the 'bgraf.inv' module, we gosub to line 1000 for the shape definition, like so:

```
1000   RESTORE
1010   FOR j=0 TO 15:FOR i=0 TO 1:READ mshape%(i,j):NEXT:NEXT
1020   RETURN
2000   DATA    0,   0
2005   DATA    0,   0
2010   DATA    0,   0
2015   DATA    0,1984
2020   DATA 1984,2080
2025   DATA 2080,4752
2030   DATA 4752,4112
2035   DATA 4112,5008
2040   DATA 5008,2080
2045   DATA 2080,1984
2050   DATA 1984,2336
2055   DATA 2976,2720
2060   DATA 1344,4752
2065   DATA    0,   0
2070   DATA    0,   0
2075   DATA    0,   0
```

Lines 1000-1020 read the contents of the data statements into the mshape% array, thereby defining the two versions of our creature. Hang on to these data statements, they will be used in several other programs later on in this article, and they are really too dull to repeat. Next comes the screen setup:

```
25   PERFORM initgrafix
30   PERFORM grafixmode(%3,%1)
35   PERFORM fillcolor(%4):PERFORM pencolor(%13)
40   PERFORM fillport
45   PERFORM viewport(%0,%139,%0,%191)
50   PERFORM grafixon
```

The lines above clear the screen to dark green and set the pen color to yellow. Note that we again use the 140X192 color mode. After this setup we initialize some general variables:

```
55   f%=4:s%=16:z%=0
60   x%(8)=-3:x%(21)=3:y%(11)=3:y%(10)=-3
65   i=70:j=90
```

Of note above is line 60, which establishes some entries into the large x% and y% arrays. A quick check of your keyboard chart should tell you that ASCII 8 is the left arrow key, 21 is the right arrow key, 11 is the up arrow and 10 is the down arrow. Left and right correspond to movement in the X direction, and up and down in the Y direction. That makes it obvious that we will be using values in the x% and y% arrays to indicate the amount and direction of movement depending on which cursor key is pressed (left and down being negative movements, respectively). As we have seen before, such techniques are wasteful of space, but increase speed. We'll see an even more interesting application of this technique in just a minute. For now, on with the program:

```
70   PERFORM moveto(%i,%j)
75   IF r%=0 THEN r%=16:ELSE r%=0
80   PERFORM drawimage(@mshape%(0,0),%f%,%r%,%z%,%s%,%s%)
85   GET a$:a=ASC(a$)
90   IF a<>27 THEN i=i+x%(a):j=j+y%(a):GOTO 70
```

The lines above constitute the main loop of the program. After moving to the position established by the initial values of i and j, r% is set to alternate between 0 and 16, which will be our bit index into the mshape% array. The 'drawimage' proceedure (from 'bgraf.inv') is then used to put the appropriate bit pattern on the screen, at the current pen location. See your Basic manual for a specific discussion of 'drawimage', but fundamentally, the parameters look like this:

```
PERFORM drawimage(@array,%Num.Row.Bytes,%X.skip,%Y.skip,%Dr.width,
%Dr.height)
```

This definition parallels, of course, the DrawBlock capability of the .Grafix driver, and allows you to specify a source array, the number of bytes in a given row of the array (needed to find the offset for row 2, etc.), the number of bits to skip in the row before drawing, the number of rows to skip in a column before drawing, the number of row bits to draw from that point, and the number of rows to draw. This is sufficient information to define any arbitrary rectangular block of bits in any given array. Any bits in the array that are 'on' (that is, 1's) are drawn in the current pencolor, and any that are 'off' (0's) are drawn in the current fillcolor.

Lines 85 and 90 get keystrokes and modify the values of 'i' and 'j' according to the contents of that character location in the 'x%' and 'y%' arrays and then jump back to 70 to redisplay the creature at the new location. If the character typed is an Escape (27), then cleanup and termination is done:

```
100   TEXT
105   PERFORM release:PERFORM release:PERFORM release
110   CLOSE:INVOKE
115   END
```

If you type nothing else in from this article, at least try the program above.

You should have some fun watching the creature swim around the screen at your command. For a little more excitement, try adding the following:

```
61   x%(55)=-3:y%(55)=3:x%(49)=-3:y%(49)=-3
62   x%(57)=3:y%(57)=3:x%(51)=3:y%(51)=-3
63   x%(52)=-3:x%(54)=3:y%(56)=3:y%(50)=-3
```

The lines above set up additional definitions of possible X and Y movements. Close examination of your ASCII chart will show that the codes correspond to numbers on the numeric pad of the Apple III. ASCII 55 in line 61 corresponds to the character '7', which is in the upper left-hand corner of the keypad. Both x% and y% are affected, x% being decremented (indicating movement to the left) and y% being incremented (indicating movement up). This combination creates diagonal motion. Quick comparisons with the rest of the characters will show the remaining relationships. Add these lines and run the program again. You'll find that you can control the creature completely from the pad! Note also that changing the constant value will change the amount of movement in any direction.

Onward, Ever Diagonally

Here's hoping that the program above has whetted you appetite for more creature features. The next program will combine creature movement with the windowing techniques of the graphics driver to create interesting motions of several creatures at once. First, however, some fooling around should be

encouraged. Try changing the displacement constants in the previous program to values higher than three. For example, try:

```
60  x%(8)=5:x%(21)=-5:y%(11)=5:y%(10)=-5
```

Now use the cursor keys. Makes a mess, right? Right. What happens is that while the previous drawblock image had enough fillcolor bits (zero value bits) surrounding the image to blank out any movement of three pixels in any direction, when we move five at a time, some old bits are left on the screen without being cleaned up by the next occurance of drawimage. A quick glance at the data statements will show that only three rows on top and three on the bottom are completely zero. Some analysis of the row values will prove that the same is true about zero bits on the left and right sides of the columns. To allow our next program some freedom as to how much displacement an image can have without leaving trash on the screen, we will do the following:

```
10   DIM mshape%(1,15),zshape%(1,31)
15   INVOKE"/basic/bgraf.inv"
20   OPEN#1,".grafix"
25   GOSUB 1000
```

The difference above is that we introduce the zshape% array, with twice as many rows as our mshape% array. This array is initalized in the routine at 1000, as follows:

```
1000   RESTORE
1010   FOR j=0 TO 15:FOR i=0 TO 1:READ mshape%(i,j):NEXT:NEXT
1020   FOR j=0 TO 15:FOR i=0 TO 1:zshape%(i,j+8)=mshape%(i,j):NEXT:NEXT
1030   RETURN
```

Please note that we use the same data statements from the last program, and, once the mshape% array is defined, we load the middle of the zshape% array with it. The offset in the rows between mshape% and zshape% gives eight extra blank rows at the top and bottom of zshape%, enough for the tricks we are about to pull.

Next, we initalize and declare a viewport, in which the visible part of our operations will occur:

```
35   PERFORM initgrafix
40   PERFORM grafixmode(%3,%1)
45   PERFORM pencolor(%4)
50   PERFORM fillcolor(%7):PERFORM fillport
55   PERFORM viewport(%40,%100,%15,%130)
60   PERFORM fillcolor(%13):PERFORM fillport
65   PERFORM grafixon
```

Note above that although the graphics routines will let us draw anywhere on the screen (and anywhere off the screen from -32768 to 32767), the only visible

effects will occur in the 40,130 to 100,15 window. Next, we get to the draw section, which is quite a bit more elaborate:

```
100   s%=16:t%=32:z%=0:f%=4
105   FOR k=0 TO 100
110     j=COS(k/5+2)*53+88
115     l=SIN(k/10)*30+58
120     m=SIN(k/10)*70+85
125     r%=s%*(r%=z%)
130     PERFORM moveto(%k+28,%j)
135     PERFORM drawimage(@zshape%(0,0),%f%,%r%,%z%,%s%,%t%)
140     PERFORM moveto(%k+14,%l)
145     PERFORM drawimage(@mshape%(0,0),%f%,%r%,%z%,%s%,%s%)
150     PERFORM moveto(%k,%m)
155     PERFORM drawimage(@zshape%(0,0),%f%,%r%,%z%,%s%,%t%)
160     NEXT k
200   GET a$:IF ASC(a$)<>27 THEN 105
```

The effect of the statements in lines 110-120 is to create different y values for each of three creature images. Lines 130-155 then move to each unique location, draw the appropriate creature image, and go on. Note that the incremental movement of the first and third shape is great enough that we need to use the zshape% version. Since zshape% takes longer to draw, mshape% is used where possible. Note also that the X positions are offset from each other, with the boundaries of the window responsible for "clipping" the images until they are within the display area. Running this program will produce images of bouncing creatures, zipping through a box-like window on the screen.

Like most programs, this one has a cleanup section:

```
300   PERFORM release:PERFORM release:PERFORM release
310   CLOSE:INVOKE
320   TEXT
330   END
```

There, have fun with that one!

By the way, if you want to edit your own creatures for the previous program, on any to follow, you can use the February Shape Editor, and make the following changes to the program above:

```
10   DIM shape%(7,15),mshape%(1,15),zshape%(1,31)
15   INVOKE"/basic/bgraf.inv","/basic/request.inv"

22   INPUT"Shape file name: ";file$
24   IF file$="" THEN 300
```

```
 28   IF error THEN PRINT"Error number "error" in file '"file$"'.":GOTO
22
 30   FOR j=0 TO 15:FOR i=0 TO 1:mshape%(i,j)=shape%(i,j):NEXT:NEXT
 32   FOR j=0 TO 15:FOR i=0 TO 1:zshape%(i,j+8)=mshape%(i,j):NEXT:NEXT

 1000   ON ERR GOTO 1030
 1010   array$="shape%":OPEN#3,file$
 1020   ftype=TYP(3)
 1030   IF ftype=1 THEN READ#3;filtyp,ch,cw,sl:IF filtyp=1 THEN 1070
 1040   OFF ERR:CLOSE#3:IF ftype=0 THEN DELETE file$
 1050   error=1:REM Not a shape file
 1060   RETURN
 1070   READ#3,1:PERFORM filread(%3,@array$,%256,@ret%):OFF ERR:CLOSE#3
 1080   IF ret%<>256 THEN error=2:RETURN:REM Shape definition is invalid
 1090   error=0:RETURN:REM Shape loaded
```

The routine from 1000 to 1090 above can be used as a general purpose shape load routine. Note the addition of the 'Request.inv' invokable to do the reading of the shape file.

Wrapping It All Up and Bouncing It Off a Wall

The program above proves that you can get the shapes to move through some rather elaborate paths. The program below puts together everything we have covered so far, and borrows from an idea in the old Applesoft Tutorial manual, that of objects (the called the little square blocks "balls") bouncing off the walls of a video room, not unlike the old "Pong" game. Since we have already shown how to endow our creations with X and Y movements, this should be easy:

```
 10   DIM mshape%(1,15)
 15   INVOKE"/basic/bgraf.inv"
 20   OPEN#1,".grafix"
 25   GOSUB 1000
```

The lines above are the usual warmup. The routine at 1000 is the usual, and uses the data statements from the previous programs, to wit:

```
 1000   RESTORE
 1010   FOR i=0 TO 15:FOR j=0 TO 1:READ mshape%(j,i):NEXT:NEXT
 1020   RETURN
```

Now on to setting up the screen:

```
 30   PERFORM initgrafix
 35   PERFORM grafixmode(%3,%1)
 40   PERFORM fillcolor(%5):PERFORM fillport
 45   PERFORM moveto(%45,%145):PERFORM pencolor(%0):PRINT#1;"Bug Box"
 50   PERFORM viewport(%40,%99,%15,%130)
```

```
55   PERFORM fillcolor(%13):PERFORM fillport
60   PERFORM viewport(%60,%80,%62,%82)
65   PERFORM fillcolor(%4):PERFORM fillport
70   GOSUB 600:REM set color table
```

After clearing the screen to gray in line 40, We print the title "Bug Box" above a window of yellow created by lines 50-55. Then a dark green square is drawn in the middle of the box by lines 60-65, and we go the the routine at 600 to set up our color table scheme:

```
600   PERFORM setctab(%7,%2,%7)
610   PERFORM setctab(%2,%7,%7)
620   PERFORM setctab(%14,%2,%13)
630   PERFORM setctab(%15,%7,%13)
640   PERFORM setctab(%14,%13,%13)
645   PERFORM setctab(%14,%7,%7)
650   PERFORM setctab(%15,%13,%13)
655   PERFORM setctab(%15,%2,%2)
660   PERFORM setctab(%2,%4,%4)
665   PERFORM setctab(%7,%4,%4)
670   PERFORM setctab(%14,%4,%4)
675   PERFORM setctab(%15,%4,%4)
690   RETURN
```

As the routine above might tend to indicate, we will employ two creatures in this demonstration, one dark blue (color 2) and one light blue (7). The dark blue creature will use a fillcolor of aqua (14) and the light blue one will use fillcolor white (15). As you can see from lines 600-610, the dark blue creature will always appear to pass behind (be covered up by) the light blue creature, should their paths cross. Lines 620-655 insure that the background (fill) colors will always translate to yellow (13), the background color of the box in which our creatures will live (and bounce). Study this carefully, until you are sure as to what is going on. Finally, lines 660-675 insure that any movement in the area of the dark green (4) box will be hidden, that is, appear to go behind that object, since all colors drawn onto color 4 will result in color 4 on the screen.

Now that our colors are set, on with the show:

```
75   PERFORM viewport(%40,%99,%15,%130)
80   PERFORM grafixon
100   f%=4:s%=16:t%=32:z%=0
105   s1x=INT(RND(1)*40)+40:s1y=INT(RND(1)*115)+15
110   s2x=INT(RND(1)*40)+40:s2y=INT(RND(1)*115)+15
115   x1m=3:y1m=3:x2m=3:y2m=3
120   ON KBD GOTO 250
```

The lines above establish a viewport for operations, and set up the initial random X and Y locations for our two creatures (lines 105-110). Line 115 sets the increment of movement in each direction for each creature, and 120 establishes an ON KBD jump to 250 to get us out of the program. We now start the loop which will bounce our creatures off the walls of their tiny domain:

```
125   x1n=s1x+x1m:IF x1n<37 OR x1n>89 THEN x1m=-x1m:GOTO 125
130   y1n=s1y+y1m:IF y1n<27 OR y1n>134 THEN y1m=-y1m:GOTO 130
135   x2n=s2x+x2m:IF x2n<37 OR x2n>89 THEN x2m=-x2m:GOTO 135
140   y2n=s2y+y2m:IF y2n<27 OR y2n>134 THEN y2m=-y2m:GOTO 140
```

The lines above add the X and Y increments (or decrements, if negative) to their respective values, and check to see if the results are within the ranges of the window. Note that the pen position is always the upper lefthand corner of the drawblock image, which explains the differences in the values from the viewport statement above. Now we get into the actual draw routines, based on the positions calculated above:

```
145   r%=s%*(r%=z%)
150   PERFORM moveto(%x1n,%y1n)
155   PERFORM pencolor(%2):PERFORM fillcolor(%14)
160   PERFORM drawimage(@mshape%(0,0),%f%,%r%,%z%,%s%,%s%)
165   PERFORM moveto(%x2n,%y2n)
170   PERFORM pencolor(%7):PERFORM fillcolor(%15)
175   PERFORM drawimage(@mshape%(0,0),%f%,%r%,%z%,%s%,%s%)
180   IF ABS(x1n-x2n)>10 OR ABS(y1n-y2n)>10 THEN 200
185   PERFORM moveto(%x1n,%y1n)
190   PERFORM pencolor(%2):PERFORM fillcolor(%14)
195   PERFORM drawimage(@mshape%(0,0),%f%,%r%,%z%,%s%,%s%)
200   s1x=x1n:s1y=y1n:s2x=x2n:s2y=y2n
205   GOTO 125
```

Notice again that we use r% to calculate the offset into the mshape% array to animate our creature. Lines 150-160 move to the calculated position of our first creature, set the colors and draw the image. Lines 165-175 do the same for the second creature. Now comes something interesting. Creature number two has priority over creature number one, if they are in the same space. Therefore drawing over number one will destroy part of the its image, which subsequent repositioning of the creatures cannot recreate. Lines 180-195 check for this possiblity and redraw creature number one, to fill in any missing parts before the next erase at the top of the loop. Try the program without these lines to see what happens. In either circumstance, line 200 sets the current positions of the creatures to the positions just used, and starts over. The effect is that of the two beasts bouncing off the four walls in quite a regular fashion. Changing the values of x1m, x2m, y1m and y2m will effect the type, length and angle of bounce. Experiment and see what you like (random numbers are fun too!)

A little more and we're finished:

```
250   OFF KBD
255   IF KBD=27 THEN 300
260   IF KBD=13 THEN POP:GOTO 40
265   ON KBD GOTO 250
270   RETURN
```

This ON KBD routine allows termination (with Escape (27)) and if RETURN is pressed, the program starts all over with new random locations for the creatures. This is useful if you want to see how the color table affects the creatures when they cross over, but the random values put them on paths that don't cross. You just press RETURN and start over with a new scenario.

Finally:

```
300   TEXT
310   PERFORM release:PERFORM release:PERFORM release
320   CLOSE:INVOKE
330   END
```

Does the usual cleanup.

A Cheerful Farewell

This month's article should give you lots to work on. As you might imagine, the color table presents all sorts of possibilities that this missive could only hint at. Next time we'll wrap up our discussion of graphics and get on to more interesting doings with the Apple III. Until then...

Exploring Business Basic - Part XXI

Two quick announcements before we begin are appropriate. First, the long awaited SOS Reference Manual and Device Drivers Writer's Guide came off the presses in April and should have reached the far corners of the earth by now. They are an incredible wealth of information about how your favorite machine really works. If anybody you know thinks SOS isn't the best single user operating system around, show him a copy! In addition to great explanations of the inner workings, the Device Drivers manual contains listings of sample disk (block) and character drivers and the SOS manual contains a diskette with a program called "EXER-SOS", designed to help debug drivers and other routines by providing a full interface to operating system and driver 'calls'. If your local dealer doesn't stock it, he should be able to easily order it for you from Apple.

The second announcement is a great one for those of you who read this article wistfully, because you have an Apple][and keep wondering what all the fuss is about concerning the Apple III, or because you need another Apple III for your business, etc. Apple dropped the price of a 256K Apple III in April to $2695! That's a full $1600 less than this time last year, and represents (in this humble Apple III owner's opinion), the best bargain in personal computers anywhere. So much for the commercial.

And Now, Back to our Regularly Scheduled Program

As was mentioned last time, this article is the last in a series exploring the text and bit map modes of graphics on the Apple III. We've covered a great deal in the last three months, from simple character set and shape definitions, through character set animation effects (including a game!!!!) and finally to color bit-mapped graphics with it's own brand of animation. That's a lot of territory, and makes it even more reasonable that the last in the graphics series (at least for now) covers a combination of the two worlds, text on the bit-mapped graphics display.

Farewell, Faithful BGRAF.INV

Those of you who have been following these articles for some time may be a bit confused by the last sentence in the paragraph above. "I know how to do text on the graphics screen", you say, "just OPEN the .GRAFIX driver and print characters to it like any other driver!". Right as rain, as they say in California, but what does that have to do with the graphics invokable in Business Basic?

If you carefully read the sections in your Basic manual on the BGRAF invokable module, and the section in the Standard Device Drivers manual on the .GRAFIX driver, you'll notice that the printing of text characters on the graphics screen is handled by the DrawBlock function of the driver, writing out the bit patterns of each character from the currently defined character set (normally the set used by the .CONSOLE driver). By using the BGRAF module, it is possible to assign a new character set to be used by DrawBlock. It is also possible to use the DrawImage function of the module to draw any shape (including text characters) onto the graphics screen. For the last couple of episodes, we've been dealing with DrawImage's shape drawing ability, to populate our screen with creatures of various shapes which crawled their way into our hearts. The marriage of the DrawImage flexibility with a text character set presents some interesting possibilities, among which is the ability to really dress up the appearance of text on the screen. It is on this intriguing capability that our final (for a while) discussion of the BGRAF invokable module will rest.

Beauty in "Proportion" to its Cost

By now many of you have probably looked at a photograph, if not an actual demo, of Apple's new Lisa system. One of the things which immediately strikes people is how pleasing the display looks. When you analyze it, it's not only the high-resolution display, but the fact that the characters are pleasingly arranged on the screen, more like text is found in a magazine or book than that found on a computer terminal. The fundamental difference in the display modes is that all text on the Lisa screen is "proportionally" spaced, that is each character is arranged to be the minimum distance from the adjoining character, not evenly spaced as in normal computer printouts and screen displays.

It's easy to see that the characters "i" and "l" don't take up nearly as much room as an "M" or "X", yet in standard computer output they do, in order that all columns will automatically line up properly, and that editing can be simplified. Such constant spacing techniques are called "Monospacing", and for certain things, like columns of numbers, they are really important. For ordinary text, however, monospacing is not only less natural and attractive, it wastes space as well. In fact, several word processors for the Apple][use the 280X192 graphics mode to create "proportional" character sets. These programs allow up to 70 characters across a screen which can accommodate only 40 monospaced characters! We will learn in this article that the higher resolution of the Apple III allows us to get over one hundred characters across the screen, with no sacrifice in readability!

Beauty on a Budget

This month's program allows you to play with the concepts of proportional spacing and text font appearance from within a program which works like a high resolution screen editor for text. The editor program permits you to place porportional or mono-spaced text anywhere on the screen, use multiple fonts on the same screen, and use the graphics driver "transfer mode" feature to do such things as overstrikes and erasures. The best feature of the program, however, is that it contains lots of routines you can use to create your own hi-res displays, both with the currently available fonts, and any of your own design. Remember that with BGRAF and the .GRAFIX driver you can have individual characters which are much larger than the normal 7X8 character cell definition. The program deals with normal size character fonts, but we'll discuss easy modifications which are possible if you want to use your own non-standard sizes. There's plenty to cover, so let's go...

```
 10   DIM
char%(63,7),cset%(511),lookup(15),flip(255),cstart%(127),clen%(127)

 15   HOME:PRINT"High Resolution Screen Character Editor":PRINT
 20   PRINT"Initializing variables, please wait"
 25   GOSUB 4000
```

The first section sets up our arrays and goes to the subroutine at 4000 to load up tables and do other initialization. For those of you who have been following this series, the arrays 'char', 'cset', 'lookup' and 'flip' should be familiar. 'Char' contain the character set of our choice and it is defined as being 128 bytes wide by 8 rows high, enough to contain the 128 ASCII character definitions in a standard font, which is 8X8 pixels per character, with the right-most pixel column used for defining flash in inverse text mode (7X8 displayable). Those of you who read the documentation will notice that a .BGRAF character set can actually contain 256 character definitions, but the standard system fonts are limited to 128, so that's what we'll go with for now. 'Cset' contains the actual image of a system font file, as read in by the "download.inv" invokable module. We'll look at a conversion routine which transforms fonts in the system format into character set definitions that BGRAF can operate on. 'Lookup' and 'flip' are used to accomplish this transformation quickly.

Which brings us to the 'cstart' and 'clen' arrays. Their size of 128 elements each should be a clue that each element contains information about the respective character associated with the element number and the names should be a clue as to what that information is. The characters in a normal character set are stored in blocks one byte wide in the 'char' array, but since we want to do proportional spacing, it is necessary to define for each character cell just how much of the character in it we want to display. In this program that is

accomplished by defining the starting pixel row within the character cell to start the display, and the number of pixel rows to display from that point. Let's look at the initialization section, where this will become more clear:

```
 4000   REM initialize
 4025   DATA 0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15
 4045   FOR i=0 TO 15:READ lookup(i):NEXT
 4055   v256=256:v512=512:v128=128:v16=16
 4060   FOR i=0 TO 255:a$=HEX$(i):flip(i)=v16*lookup(TEN(MID$(a$,4,1)))
        +lookup(TEN(MID$(a$,3,1))):NEXT
```

The section from lines 4000 to 4060 above is identical to routines in previous programs. It creates the 'flip' array, which is simply a lookup table of all the possible combinations of the value of a byte of data (256 in all) with their values if the bits in the byte were exactly reversed, that is, flipped over. This table is required by the routine which transforms a character set from system font format to .GRAFIX format. Apple cleverly made these two formats the exact reverse of each other. Next comes more initialization of variables:

```
 4070   name$="Font":array$="cset%":size%=1024
 4085   ctrl$=CHR$(8)+CHR$(21)+CHR$(11)+CHR$(10)+CHR$(13)+CHR$(27)
 4090   left=0:right=559:top=191:bot=18:cvert=191:chorz=0:
delay=1:noset=1
 4100   hspace=7:vspace=8:reverse=0
 4110   bell$=CHR$(7):eras$(0)="Replace":eras$(1)="Overlay":
eras$(2)="Invert "
 4120   eras$(3)="Erase  ":prop$(0)="Monospace  ":
prop$(1)="Proportional"
```

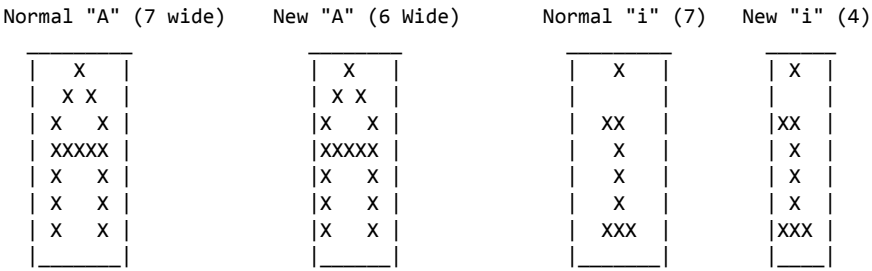
Several variables are worthy of note above. First, in line 4085 a string is built of command characters which will be used later for fast lookup by the editor command routine. In order, the characters are back arrow, forward arrow, up arrow, down arrow, carriage return and escape. Line 4090 sets some initial constraints for the area which may be edited, and sets the initial position of the cursor to 0,191 (chorz and cvert). Line 4100 sets the default for horizontal and vertical character spacing. These are values you would want to change if you used other, non standard character sets. Next comes a big table which will be a challenge to type in:

```
 4200   DATA 3,4,3,2,2,4,1,6,1,6,1,6,1,6,3,2
 4205   DATA 1,4,3,4,1,6,1,6,2,3,2,5,3,2,1,6
 4210   DATA 1,6,1,6,1,6,1,6,1,6,1,6,1,6,1,6
 4215   DATA 1,6,1,6,3,2,2,3,1,5,2,5,2,5,1,6
 4220   DATA 1,6,1,6,1,6,1,6,1,6,1,6,1,6,1,6
 4225   DATA 1,6,2,4,1,6,1,6,1,6,1,6,1,6,1,6
 4230   DATA 1,6,1,6,1,6,1,6,1,6,1,6,1,6,1,6
 4235   DATA 1,6,1,6,1,6,2,5,1,6,2,5,1,6,1,6
```

```
4240   DATA 2,4,1,6,1,6,1,6,1,6,1,6,1,6,1,6
4245   DATA 1,6,2,4,1,5,1,6,2,4,1,6,1,6,1,6
4250   DATA 1,6,1,6,1,6,1,6,1,6,1,6,1,6,1,6
4255   DATA 1,6,1,6,1,6,1,6,2,3,1,6,1,6,0,7

4300   FOR i=32 TO 127:READ cstart%(i),clen%(i):NEXT
4500   RETURN
```

Notice that there are eight sets of two values on each line and 12 lines, for a total of 96 sets of values, one for each of the "displayable" characters in a font definition. To save trouble and improve performance, we have not bothered to make the control character definitions displayable, so the characters from 32 ('space') through 127 ('DLE') are the only ones defined. Notice that most characters seem to follow the pattern '1,6' meaning "start one pixel over in the character definition and use the next 6 pixels". This is because a large number of characters in the standard set are five pixels wide, and we need one pixel for spacing. For consistancy, we always steal the extra pixel from the leading edge of the character, leaving the spacing pixel at the end. This does mean, however, that we save at least one pixel per character, which for 80 characters across on a normal screen is 80 pixels, or the equivalent of an extra 13 characters from that technique alone! Six wide monospacing thus buys the ability to write 93 characters across the screen and then, depending on the number of "i's" and "l's" and other skinny characters that get used, the actual total may be much higher. An example would help:



Notice that the new "A" is obtained by simply shaving off the first column which would read "1,6" in our "starting pixel, character width" format. In the case of the lower case "i", the formula is "2,4" since we are to skip the first two pixel columns and then only use four to draw the character.

In any case the table presented in lines 4200 to 4255 will define the slim new format for all the characters in the font we will be using. Note that this definition is for the "Standard" font. Several other fonts come with the Business Basic disk, such as "Roman", "Apple", etc. These fonts contain characters which are designed a little differently than the standard font characters, so you should construct a different table for those characters.

Well, having beaten initialization to death, let's get on with the show:

```
 30   INVOKE"/basic/bgraf.inv","/basic/request.inv","/basic/
download.inv"
 35   OPEN#1,".grafix"
 50   INPUT"Use Normal or Inverse screen (N or I)? ";a$
 55   IF a$="" THEN GOTO 1050
 60   a$=MID$(a$,1,1):op=INSTR("NnIi",a$)
 65   IF NOT op THEN 50:ELSE op=INT((op+1)/2)-1
 70   fill=op*15:pen=( NOT op)*15
 75   erasenum=2+4*op
```

Because we'll be dealing with fonts as well as the .GRAFIX driver, we invoke the three modules listed in line 30, all of which are found on the Business Basic master diskette. Then the graphics driver is opened, allowing us a path to print text commands on the graphics screen and perform other functions.

Line 50 through 75 then request whether the screen is to be displayed in normal mode (white characters on a black background) or inverse mode (just the opposite). Notice the use of the INSTR statement and the calculations in lines 60 and 65 to check for mismatches and then set 'op' to 0 or 1 for normal or inverse. Line 70 then sets 'fill' and 'pen' to either 0 and 15 or 15 and 0, depending on the value of 'op'. An IF statement would work just as easily, but practice in logical value calculations can't hurt.

The variable 'erasenum' in line 75 is used to define what transfer option is appropriate to use to draw the cursor so that it doesn't interfere with the text which will be on the screen. If the pencolor is 0 (black) then transfer option 2, "invert", is used. If the pencolor is 15 (white) then option 6 is used, "inverse invert". More detail on how these transfer options work will be found later in this article. Next comes more initialization:

```
 100   PERFORM initgrafix
 105   PERFORM grafixmode(%2,%1)
 110   PERFORM fillcolor(%fill):PERFORM pencolor(%pen)
 115   HOME:PRINT:PRINT"Initializing the graphics screen, please wait."
 120   erase=0:prop=1
 130   GOSUB 3600
```

In line 105 the program sets mode 2, the 560X192 mode, and 110 uses the previously defined fill and pencolor variables. Line 120 sets the initial values for 'erase' which defines the transfer option used to write characters onto the screen, and 'prop' which defines whether proportional or monospaced character writing will be performed. Then the subroutine at line 3600 is called to set up the screen:

```
 3600   PERFORM viewport(%0,%559,%0,%191):PERFORM fillport
 3620   PERFORM moveto(%0,%9):PERFORM linerel(%559,%0)
```

```
 3630   PERFORM moveto(%412,%9):PERFORM linerel(%0,%-9)
 3640   IF reverse THEN RETURN
 3650   warn=1:message$=eras$(erase)+" "+prop$(prop):GOSUB 3100
 3660   RETURN
```

Lines 3600-3630 clear the screen to the current fill color and then create a message area in the bottom ten lines of the screen. If 'reverse' is on (more on that later), then the operation is finished. Otherwise, the subroutine at line 3100 is called to write the current state of the transfer option and the spacing mode:

```
 3100   PERFORM xfroption(%0)
 3105   GOSUB 3500
 3110   PERFORM moveto(%7+412*warn,%7):PRINT#1;message$;
 3120   IF NOT warn THEN FOR i=1 TO 750*delay:NEXT
 3125   PERFORM xfroption(%( NOT( NOT erase))*(erasenum-2+erase))
 3130   IF NOT warn THEN GOSUB 3500:ELSE:warn=0
 3135   RETURN

 3500   PERFORM viewport(%5+408*warn,%411+148*warn,%0,%8):PERFORM
fillport
 3510   PERFORM viewport(%0,%559,%0,%191)
 3520   RETURN
```

The subroutines above give the program a general purpose ability to write messages to the user on the bottom of the screen. Line 3100 above first sets transfer option 0, called "replace", which will overwrite any message currently in the message area. Then subroutine 3500 is called to clear the appropriate part of the message area. If 'warn' mode is on, then the message is written from horizontal positions 413 to 559, between rows 0 and 8 on the screen, the lower righthand corner. If 'warn' is zero, then the message is written between positions 5 and 411. These techniques can be used to establish several message "windows" if necessary, each using the same routines.

Line 3110 then moves to the appropriate place on the screen and writes the message in 'message$'. Messages in the "warning" window remain until changed by another message, but if the regular message window is used, the program pauses briefly (line 3120) and then sets the transfer mode back to its original state and calls line 3500 to clear the window.

Back Home from the Subroutines

Now that the program has set up the screen and the default modes are established, we turn on the screen and start using the program:

```
 150   PERFORM grafixon
 155   message$="High Resolution Screen Editor":GOSUB 3100
 160   GOSUB 1400
```

```
 165   IF fin=2 THEN 1000:ELSE IF fin AND noset THEN 1000
 170   GOSUB 3500
```

After putting an identifying message on the bottom of the screen, the section above uses the subroutine at line 1400 to load the character font which will be used to write on the screen. That routine starts like this:

```
 1400   prompt$=name$+" pathname: "
 1425   GOSUB 3000:IF fin THEN RETURN
```

The font subroutine above first prompts for the pathname of the font file, by using the variable prompt$ and the high-resolution input routine at line 3000 (more on that routine in a minute). The variable 'fin' in line 1425 is used to indicate a non-standard exit from input, such as pressing "escape" or a carriage return at the beginning of the input. In any case, the font name, if any, is returned in the variable 'line$'. A quick look at the input routine is appropriate now (are you beginning to feel like the Apple III itself, plunging deep into subroutines again?):

```
 3000   PERFORM xfroption(%0)
 3005   GOSUB 3500:PERFORM moveto(%7,%7):PRINT#1;prompt$;
 3010   line$="":fin=0
 3015   GET a$:a=ASC(a$):IF a<32 THEN 3030
 3020   IF LEN(line$)<40 THEN PRINT#1;a$;:line$=line$+a$:ELSE:PRINT
bell$;
 3025   GOTO 3015
 3030   IF a=13 THEN fin=(LEN(line$)=0):GOSUB 3500:GOTO 3070
 3035   IF a=27 THEN fin=2:GOSUB 3500:GOTO 3070
 3040   IF a=24 THEN PRINT bell$;:GOTO 3005
 3045   IF a<>8 THEN 3015
 3050   IF LEN(line$)=0 THEN 3015
 3055   PERFORM moverel(%-7,%0):PRINT#1;" ";:PERFORM moverel(%-7,%0)
 3060   line$=MID$(line$,1,LEN(line$)-1)
 3065   GOTO 3015
 3070   PERFORM xfroption(%( NOT( NOT erase))*(erasenum-2+erase))
 3075   RETURN
```

The routine above is very similar to others in this column which accept input while on the high-resolution screen. It supports typing in characters, and erasing input with the back-arrow. In addition, a check is made in line 3020 to be sure that the input doesn't overflow past 40 characters and thus into the next message window. If everything's ok, the character is added to 'line$', and the routine loops back up to get the next character.

Control character processing, including back arrow erasing, is handled in lines 3030-3065. Backing up is a matter of moving the cursor location back one space, printing a blank, and then moving back to accept the new character. In

addition, line 3060 removes the previous character from the end of 'line$', and then loops back to get the next character. Lines 3030-3040 handle special character exits from the routine. A carriage return exits with 'fin' set to 0 if there are characters in line$, or 1 if the string is null. A value of 2 for 'fin' indicates that ESCape has been pressed, and a "control-X" erases the input line, just as it does in Basic, except that the prompt is redisplayed. In all, this is a useful routine which could be easily adapted for use anywhere on the screen.

Now that we have a general way to get input, and to check for certain control characters like ESCape, we continue our look at the font request routine:

```
1435   ON ERR GOTO 1460
1440   font$=CHR$(34)+line$+CHR$(34):PERFORM getfont(@font$,@array$)
1445   OFF ERR:GOSUB 2000
1450   message$=name$+" loaded.":GOSUB 3100:GOSUB 2100
1455   RETURN
```

The routine above first sets up an ON ERRor jump to line 1460 and then tries to use the 'getfont' proceedure from the "download.inv" module to load a font from the file specified in 'line$'. If anything goes wrong, for example, the file specified is not a font file, then a jump is made to the error routine:

```
1460   ON ERR GOTO 1490
1465   OPEN#3,line$
1470   PERFORM filread(%3,@array$,%size%,@ret%)
1475   OFF ERR:CLOSE#3
1480   IF ret%=size% THEN GOSUB 2000:GOTO 1450
1485   message$=name$+" in "+line$+" is invalid.":GOSUB 3100:GOTO 1400
1490   message$="Not a "+name$+" file.":GOSUB 3100
1495   OFF ERR:IF TYP(3)=0 THEN CLOSE#3:DELETE line$:ELSE:CLOSE#3
1500   GOTO 1400
```

The routine at 1460 attempts to open the file as a Basic file, and use the 'filread' procedure from "request.inv" to read a font definition array. This might be the case if you used the Character and Shape Editor from a previous article, or some other means to get a font definition into a file without being able to change the file type to "FONT". If something goes wrong this time, then the appropriate error message is displayed, either in line 1485 if the file could be opened and read from but contained wrong information, or line 1490 if there was trouble opening or reading the file. The TYP function in line 1495 serves to check if the file is blank because the original OPEN caused it to be created. If so, it is deleted and control goes back to the beginning to ask for the filename again.

In the hopeful event that the filename is correct and the program can read a font from it, a subroutine at line 2000 is called to prepare the font for use by the program (deeper, ever deeper into the routine abyss):

```
2000   message$="Preparing the character font.":GOSUB 3100
```

```
 2010   FOR k=0 TO 511:b$=HEX$(cset%(k)):cset%(k)=TEN(HEX$(v256*
        flip(TEN(MID$(b$,1,2)))+flip(TEN(MID$(b$,3,2))))):NEXT
 2020   RETURN
```

The routine above has also been seen in previous episodes, since the standard system font definitions used by text mode are exactly the reverse (on a character basis) from those of the high-resolution DRAWBLOCK fonts. Line 2010 uses the 'flip' array to define all possible byte values and their corresponding reversed values. The routine is compacted on one line for maximum performance. Now that the font is flipped, it is necessary to expand it from the single dimensioned array 'cset%' to a form more readily usable by "Drawimage":

```
 2100   message$="Transferring Font format to Character set
format":GOSUB 3100
 2105   FOR k=16 TO 63:j=8*k-1
 2110     FOR i=0 TO 7 STEP 2:j=j+1:a$=HEX$(cset%(j)):b$=HEX$(cset%
(j+4))
 2115       char%(k,i)=TEN(MID$(a$,1,2)+MID$(b$,1,2))
 2120       char%(k,i+1)=TEN(MID$(a$,3,2)+MID$(b$,3,2))
 2125       NEXT:NEXT
 2140   message$="Font format transferred":GOSUB 3100:GOSUB 3500
 2150   noset=0
 2160   RETURN
```

Note above that we are unpacking 'cset%' into a simple row, column format in the array 'char%, where rows represent rows of bits, and columns represent character definitions, two per integer value. More information on this format can be found in previous articles in this series, and in the Business Basic manual, volume 2, under the discussion of the BGRAF invokable. Note also that you could speed up this process considerably by pre-storing a font definition in a file as a character set array in the 'char%' format. That would require modification of the subroutine at 1400 which reads in the array, but would allow quick switching of fonts without the complicated preparation routines.

Now for Something Completely Useful

Those of you who have been following along with our nested subroutine calls above will now realize that we are back to the main routine, with a font definition loaded in the 'char%' array. That brings us to the interesting stuff, actually putting some of these characters onto the screen using the proportional spacing tables. Our first step is to put a "cursor" on the screen, to indicate where the character will be written:

```
 200   PERFORM xfroption(%erasenum)
 205   PERFORM moveto(%chorz,%cvert)
 210   PERFORM drawimage(@char%(0,0),%v128,%995,%0,%1,%vspace)
```

```
215   GET a$
220   PERFORM drawimage(@char%(0,0),%v128,%995,%0,%1,%vspace)
```

First we set the transfer option for "invert". This allows the cursor to overwrite information on the screen by turning black to white, and white to black. Line 205 positions the cursor to the current horizontal and vertical value, and line 210 writes the cursor on the screen using the "drawimage" routine. This statement draws a vertical bar of bits from the 'char%' array which is one pixel wide and 8 pixels ('vspace') high starting at bit location 995 in the character set. It happens that 995 is the location in the standard character font of the "vertical bar" character (ASCII 124). By multipling 128 by 8 (the width of a character definition) and adding 3 for the offset within the cell for the bar, we obtain 995. Thus our cursor consists of a vertical bar at the extreme lefthand edge of the current character cell. This is a convenient definition of a proportional cursor, since we will be writing characters with many different widths, and we would like to know exactly where the character will be placed. If you use different fonts, you may well want to define another character position to contain this cursor character, and change the values in lines 210 and 220.

In line 215 we get a character from the keyboard, and immediately print the cursor again, exactly on top of the previous one. Since we are in "invert" mode on the transfer option, this second printing of the cursor simply changes everything back to its original state.

Now it's time to put this typed character onto the screen:

```
225   PERFORM xfroption(%( NOT( NOT erase))*(erasenum-2+erase))
230   key=ASC(a$):skp=1:IF key<32 OR key>127 THEN 270
240   IF prop THEN hspace=clen%(key):xskip=key*8+cstart%(key):
      ELSE:hspace=7:xskip=key*8
245   IF right<chorz+hspace THEN PRINT bell$;:GOTO 200
250   PERFORM drawimage(@char%(0,0),%v128,%xskip,%0,%hspace,%vspace)
255   chorz=chorz+hspace
260   GOTO 200
```

Notice that first we put the screen back into whatever transfer mode was selected, and then set 'key' equal to the ASCII value of the typed character. A check is made to see if the character is a control character or if "open-Apple" was pressed along with the character (key>127). If not, then the character is a printing one which is to be put on the screen. Line 240 sets 'hspace' and 'xskip' depending on whether proportional or monospace mode is being used. 'Hspace' indicates how wide is the character to be written; 7 for monospace or the value in 'clen%' if proportional. 'Xskip' is the offset in the 'char%' array at which to start the transfer of the character. It's always the character code times eight for monospace, with the additional value 'cstart%' if proportional. Once the correct values are computed, a check is made to see if the result will go past the right edge of the window defined by 'right'. If everything is ok then the

character is written by line 250 and the current horizontal position is updated. Then a loop back to the beginning is done to redisplay the cursor and start the process over.

As was mentioned before, control and other special characters are handled by the routine at line 270:

```
270   IF key=27 THEN 400
275   IF key>127 THEN skp=0:key=key-128
280   kvl=INSTR(ctrl$,CHR$(key))
285   ON kvl GOTO 340,350,360,370,380,160
290   GOTO 200
340   hmove=chorz-skp*(hspace-1)-1:IF left<=hmove THEN chorz=hmove
345   GOTO 200
350   hmove=chorz+skp*(hspace-1)+1:IF right>=hmove THEN chorz=hmove
355   GOTO 200
360   vmove=cvert+skp*(vspace-1)+1:IF top>=vmove THEN cvert=vmove
365   GOTO 200
370   vmove=cvert-skp*(vspace-1)-1:IF bot<=vmove THEN cvert=vmove
375   GOTO 200
380   chorz=0
385   IF bot<=cvert-vspace THEN PRINT#1;CHR$(10);:cvert=cvert-vspace
390   GOTO 200
```

First a check is made in line 270 to see if the character typed was an "ESCape". If so, an immediate jump is made to line 400 to do processing of commands. If not, a check is made for "open-apple" in line 275, and if so, the skip flag is set to zero, indicating that cursor positioning is to be done in one pixel increments. The character typed is then checked against values in 'ctrl$' which was defined on line 4085. It was defined a while back so let's reproduce it for quick study:

```
4085   ctrl$=CHR$(8)+CHR$(21)+CHR$(11)+CHR$(10)+CHR$(13)+CHR$(27)
```

The first four characters represent the cursor arrows back, forward, up and down. The fifth is the carriage return, and the sixth is ESCape. Since we have already checked once for escape in line 270, ESCape can only be detected at line 280 if it was "open-apple ESCape". Similarly, the "open-apple" key can be used with the cursor keys to signal single pixel or normal movement in a given direction. This also allows you to define additional commands, either control characters or printing characters with "open-apple" on, to implement other features in the program. The routines themselves from 340 to 390 are pretty straight-forward. Cursor movements are first checked against the 'left', 'right', 'top' and 'bot' limits, and then the cursor position is updated. Remember that when the program redraws the cursor it first positions it to the current values of 'chorz' and 'cvert' in line 200, which means there is no need to redraw the cursor in these routines.

Next let's look at the routines at line 400, which processes requests for command and mode settings:

```
 400   prompt$="Proportional or Monospace characters: ":GOSUB 3000
 405   GOSUB 3500:IF fin=2 THEN 200:ELSE IF fin THEN 425
 410   a$=MID$(line$,1,1):a=INSTR("PpMm",a$):IF a THEN a=INT((a+1)/2)
 415   ON a GOSUB 520,530
 420   message$=eras$(erase)+" "+prop$(prop):warn=1:GOSUB 3100

 520   prop=1:hspace=6:RETURN
 530   prop=0:hspace=7:RETURN
```

The request is made in line 400, and 'a' is given the value 0 (mismatch), 1 (if proportional) or 2 (if monospace). The subroutine sets the appropriate flag, and the default spacing to be used. Then line 420 takes care of writing the latest status in the "warning" window.

Next, the transfer option value is requested:

```
 425   prompt$="Replace, Overlay, Invert or Erase mode: ":GOSUB 3000
 430   GOSUB 3500:IF fin=2 THEN 200:ELSE IF fin THEN 450
 435   a$=MID$(line$,1,1):a=INSTR("RrOoIiEe",a$):IF a THEN a=INT((a+1)/
2)
 440   ON a GOSUB 500,505,510,515
 445   message$=eras$(erase)+" "+prop$(prop):warn=1:GOSUB 3100

 500   erase=0:RETURN
 505   erase=1:RETURN
 510   erase=2:RETURN
 515   erase=3:RETURN
```

The settings of "Replace", "Overlay", "Invert" or "Erase" correspond to the descriptions in the Business Basic manual on the XFROPTION routine of BGRAF.

Remember that the actual transfer option value used will depend on whether the screen is in inverse or normal mode. The routines at 500-515 may look too simple to make into subroutines, but using this structure allows you to easily add enhancements to the options. Now, on with more commands:

```
 450   prompt$="Normal, Inverse, Clear or Reverse: ":GOSUB 3000
 455   GOSUB 3500:IF fin THEN 200:ELSE:a$=MID$(line$,1,1)
 460   a=INSTR("NnIiCcRr",a$):IF a THEN a=INT((a+1)/2)
 465   ON a GOSUB 540,560,580,590
 470   GOTO 200

 540   fill=0:pen=15:erasenum=2
```

```
550   GOTO 565
560   fill=15:pen=0:erasenum=6
565   PERFORM fillcolor(%fill):PERFORM pencolor(%pen)
570   RETURN
580   GOSUB 3600
585   RETURN
590   reverse=1
595   SWAP pen,fill:PERFORM pencolor(%pen):PERFORM fillcolor(%fill)
600   PERFORM xfroption(%erasenum):GOSUB 3600
610   IF erasenum=2 THEN erasenum=6:ELSE erasenum=2
615   reverse=0
620   RETURN
```

The same structure is used to get command values, but this time the command implementation is a bit more complicated. The inverse and normal commands are handled in 540-570 by setting the appropriate values of 'fill' and 'pen' colors, and then using the BGRAF routines to pass those values to the driver. Clear simply calls the screen setup routine at line 3600 to redisplay everything, and in the process, erase everything on the current screen.

The "Reverse" command in lines 590-620 works by changing the pen and fill colors and then setting invert mode (which changes white to black and black to white) and then clearing the screen with the same subroutine at 3600 that was used to erase the screen in the previous command. The result is really nice to watch, as a "curtain" rolls down the screen, inverting everything it encounters. Think how tough that would be to do if you had to examine every pixel yourself and decide what to do!

Farewell to the Fun Stuff

Well, that about wraps up the commands to implement and the main routine which displays the characters in the various modes. Now it's time to wrap up the program with the routine at line 1000:

```
1000   REM clean up and go home
1005   prompt$="Quit the Screen Editor? (Y to confirm): "
1010   GOSUB 3000
1015   IF NOT INSTR("Yy",line$) THEN GOTO 170
1020   HOME:TEXT
1025   PERFORM release:PERFORM release:PERFORM release
1030   INVOKE
1035   CLOSE
1040   END
```

That's it! Have fun experimenting with the various modes. Especially try to type a line of proportional text and then a line of the same text in monospace below

it. You'll find an average of 20 to 30 percent more characters per line can be written, and still be extremely easy (and some would say, more pleasant) to read. Be sure to also try the effect of the various transfer options on your ability to write to the screen. Since you can position the cursor to exact pixel locations, you can also have fun with superscripts, subscripts, underlining, bold, and other effects. Be sure also to try other character fonts, like Roman and Apple. The proportional mode table will not work exactly right, but will do well enough to show the effect.

Whats's Next

This program is one of those which you could almost infinitely enhance. One immediate thought is that you will want to use GSAVE and GLOAD to store and retrieve your screen images, especially if you have a program to print screen images to a printer. This would allow you to make copies of your edited screens. You would probably want to clear the message line before saving the screen, however.

Other suggestions would include the ability to store several fonts in memory at once, and switch quickly between them for various effects. You would also have to store proportion tables for each one. If you have a font editor, you may want to create a special proportional font definition, with narrower characters than those found in the standard font. This would enable you to put many more than the 100 characters across possible with the regular fonts. Another use would be to modify the character definitions to allow larger than normal fonts for such things as gothic, shadow and other uses.

With enough work the program could be a general screen design package, with the line and circle editor from Article 8, combined with with general character capabilities of this article.

As you can see, programming, like papers on a desktop or appointments in a day, has the ability to expand infinitely to fill up any available space and time. Use yours wisely, and come back next time for some new challenges!

Exploring Business Basic, Part XXII

Last time we wrapped up our discussion of graphics (for a while) and envisoned some more practical activities with which to spend our programming hours. Somehow "bug-mania" and the bouncing "squid brothers" seemed frivolus at the time, but definitely fun. In this article we will put together some of the tricks we learned in the graphics world to demonstrate a really superior input environment, which your can use as a general purpose data entry routine.

Getting Some Utility from Basic

Everybody who has used the Apple III has had occasion to use the System Utilities program. This is the program which comes with every system and handles the tasks of formatting disks, copying files, and configuring drivers (among a million other tasks). As was said, lots of people have used this program, but not nearly as many have used one of it's special capabilities, one which we will copy extensively for this article. This function is the "insert" mode in input fields within the program. Try the following:

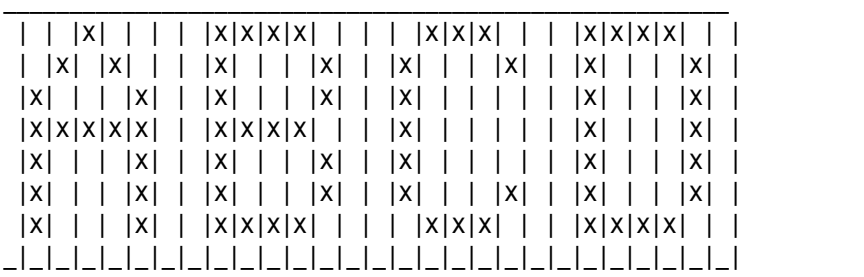
Boot System Utilities and select something simple like listing a file. When the default file name appears in the filename field at the bottom of the screen, press "open-apple I" to put the field into "insert" mode. You will immediately see a strange looking shape at the beginning of the field, resembling an upside-down "T". It looks something like this: _|_ . By moving this new cursor, you can place it actually between two characters in the input field. If you watch carefully, you will see that the characters appear to "ripple", that is quickly expand and contract, as you pass over them with this new cursor. Any characters typed when this cursor is on will appear to the right of the cursor location, inserted between the two screen characters that were divided by the cursor. Using "open-apple <--" and "open-apple -->" you can delete characters to the left or to the right of the cursor, respectively.

All in all, this "insert mode" editing is quite useful, especially since it is very clear where characters are to be inserted and deleted (not always obvious in Applewriter!). For more information as to how this mode works, consult the owners guide, or press "open-apple ?" for a quick menu of features.

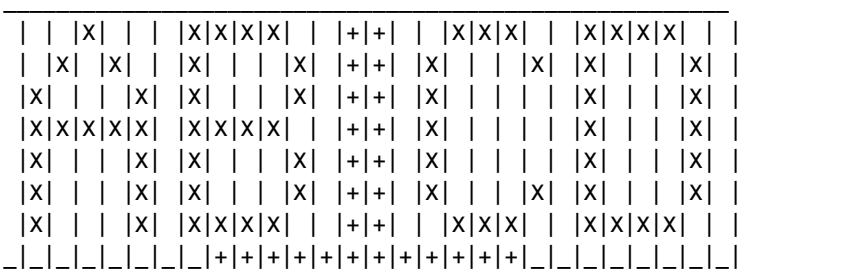
Shift to the Left, Shift to the Right...

The rippling effect is a clue to how the insert mode editing works. Normally there is one row of blank pixels on each side of a character definition. That is, the normal character cell is seven pixels wide by eight high, but normally the characters only use five horizontal pixels and seven vertical ones (the bottom row is reserved for characters with descenders, like "g" and "p"). We used this

fact in the last article to build a proportional spacing editor by squeezing out the extra blank columns and thereby packing the characters more tightly on the screen. By packing the characters more tightly in text mode, we can create enough space between them to put a vertical cursor line, and thereby indicate exactly where the insertion of a character will occur. The following example will summarize the possibilities:



The diagram above shows how the letters "ABCD" are presented on the screen as 5 by 7 character definitions within a 7 by 8 character cell. Notice that there are two blank rows between adjacent characters. If we could squeeze two of the characters apart slightly, there would be room to put a cursor between them, like this:



Notice that all the letters are legible and separated, but the "B" is moved to the left and the "C" is moved to the right to make room for the cursor in the middle. This leaves one space between "A" and "B" and between "C" and "D", but everything works out.

Getting There is Half the Fun

Now that we know that we can put a cursor between text mode characters without destroying the character definitions, the next big trick is figuring out how to accomplish this task. A solution to this problem was described in several previous episodes, and envolves creating different character definitions of each character, one definition with the "T" shaped cursor on the left, and one on the right. Defining each character in this way will allow us to see how the

compressed characters work. Fortunately, we don't have to start character definitions from scratch. The following program shows how a given character font can be transformed into the compressed set with cursors installed:

```
 10   DIM
highr%(15),lowr%(15),carryr%(15),highl%(15),lowl%(15),carryl%(15)
 20   DIM sleft%(255),sright%(255),char%(511),charl%(511),charr%(511)
 30   INVOKE"/basic/request.inv","/basic/download.inv"
 40   GOSUB 4000
```

The arrays in line 10 above are used by the conversion routine which we will see in a minute, as are 'sleft%' and 'sright%'. 'Char%', 'charl%' and 'charr%' are used to store font definitions, for regular, left-shifted and right-shifted, respectively. Line 30 invokes the 'request' module, which will be used to make control calls to SOS, and 'download' which puts the converted fonts into the system character set. The GOSUBs to line 4000 is where all the excitement starts in the character conversion process:

```
 4000   DATA 0,2,4,6,0,2,4,6,8,10,12,14,8,10,12,14
 4010   DATA 1,3,5,7,9,11,13,15,1,3,5,7,9,11,13,15
 4020   DATA 0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1
 4030   DATA 4,4,5,5,6,6,7,7,12,12,13,13,14,14,15,15
 4040   DATA 0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7
 4050   DATA 0,8,0,8,0,8,0,8,0,8,0,8,0,8,0,8

 4100   FOR i=0 TO 15:READ highl%(i):NEXT:FOR i=0 TO 15:READ lowl%
(i):NEXT
 4110   FOR i=0 TO 15:READ carryl%(i):NEXT
 4120   FOR i=0 TO 15:READ highr%(i):NEXT:FOR i=0 TO 15:READ lowr%
(i):NEXT
 4130   FOR i=0 TO 15:READ carryr%(i):NEXT
```

Each constant in rows 4000 to 4050 represents the shifted version of one nibble (4 bits) of an 8 bit row definition of a character. The exceptions are rows 4020 and 4050, which contain the quantity to carry in the event that a shift right or left needs to shift a value into the adjacent nibble. The nibble versions of this shift, along with the carry array are used to simplify the problem of having all 256 combinations of shifts of a byte in one big table. With the individual nibble shifts, it is easy to construct the table:

```
 4150   FOR i=0 TO 15:FOR j=0 TO 15
 4160       sleft%(i*16+j)=16*(highl%(i)+carryl%(j))+lowl%(j)
 4170       sright%(i*16+j)=16*highr%(i)+lowr%(j)+carryr%(i)
 4180       NEXT:NEXT
```

Lines 4150-4180 put the nibbles together and add the carry to build the conversion arrays 'sleft%' and 'sright%'. These are the constants which a

character definition will utilize to perform the actual shift. To use the tables to translate a character font, we first need one:

```
4200   prompt$="Character font pathname: ":GOSUB 5000
4210   IF error THEN RETURN

5000   PRINT prompt$;:INPUT"";a$
5010   IF a$="" THEN error=1:RETURN
5020   error=0:RETURN

4220   ON ERR GOTO 4260
4230   font$=CHR$(34)+a$+CHR$(34):charset$="char%"
4240   PERFORM getfont(@font$,@charset$)
4250   OFF ERR:PRINT"Font loaded":GOTO 4400
4260   ON ERR GOTO 4300
4270   OPEN#1,a$:PERFORM filread(%1,@charset$,%1024,@ret%)
4280   IF ret%=1024 THEN OFF ERR:GOTO 4400
4300   OFF ERR:PRINT a$" is not a valid character font file"
4310   IF TYP(1)=0 THEN CLOSE#1:DELETE a$:GOTO 4200
4320   CLOSE#1:GOTO 4200
```

The routine above is yet another variation on the familiar theme of loading in a font definition from a regular font file, or from a file created by a font editor which cannot change the file type to system type "font". When it exits to line 4400, the font is contained in 'char%', which is then sent to line 4400 for the actual conversion:

```
4400   PRINT:PRINT"Preparing the character fonts"
4410   FOR i=0 TO 511 STEP 4
4420     FOR j=0 TO 2:a$=HEX$(char%(i+j))
4430       l=TEN(MID$(a$,1,2)):r=TEN(MID$(a$,3,2))
4440       charl%(i+j)=TEN(MID$(HEX$(sleft%(l)),3,2)+
           MID$(HEX$(sleft%(r)),3,2))
4450       charr%(i+j)=TEN(MID$(HEX$(sright%(l)),3,2)+
           MID$(HEX$(sright%(r)),3,2))
4460       NEXT j
4470     a$=HEX$(char%(i+3)):l=TEN(MID$(a$,1,2))
4480     charl%(i+3)=TEN(HEX$(sleft%(l))+"7F")
4490     charr%(i+3)=TEN(HEX$(sright%(l))+"FE")
4500     NEXT i
4510   RETURN
```

The routine above deserves some careful study. What's happening is that the font definitions of each character are stored in four consecutive integer values in the 'char%' array. The routine converts the quantity to a hex value, then splits it into two bytes, and passes those through a conversion and then a lookup of the

corresponding value in 'sright%' or 'sleft%' depending on the desired shift. Lines 4470-4490 take care of the special case of the underline (part of the bottom of the upside down "T"), by forcing the line to contain all ones, no matter what character information was there. Now that we have definitions in 'charl%' and 'charr%', we can proceed with the fun:

```
 50   PRINT"Press ESCAPE to switch fonts, RETURN to exit"
 60   array$="char%"
 70   start=0
 75   PERFORM loadfont(@array$)
 80   GET a$
 90   IF a$=CHR$(27) THEN 200
100    IF a$<>CHR$(13) THEN start=start+1:GOTO 120
110    IF start=0 THEN 300:ELSE:start=0:PRINT:GOTO 80
120    PRINT a$;:GOTO 80
200    IF array$="charl%" THEN array$="charr%":ELSE:array$="charl%"
210    GOTO 75
```

After loading up the normal font in line 75, the routine accepts a character and checks it for "escape". If so, the character set is toggled between left and right shifted characters by re-loading the new set in line 75. Otherwise, the program allows you to type characters to observe them in this new definition. A carriage return as the first character on the line gets you out to line 300:

```
300   PERFORM loadfont(@charset$)
305   STOP
310   INVOKE
320   END
```

One thing you will notice quickly is that the screen in this program is ugly. Because each character, no matter what, is shifted and bordered by an L-shaped piece of the cursor, the whole screen appears to be made up of a grid in which you type letters. This is true even for spaces, since they get converted too. In a minute we'll see how to take these capabilities and make an attractive field editor out of them, but for now you can remove the grid lines from the space characters by inserting this line in the program above:

```
4505   REM FOR i=128 TO 131:charl%(i)=0:charr%(i)=0:NEXT
```

This will blank out the "space" code, and make spaces appear as normal blanks.

Two for "T"

Our objective when we began this was to figure out a way to just change the two characters surrounding the cursor (our upside down "T"), and thereby make a lot clearer job of what's going on in the editing process. The example above

was designed to show that we really could convert character sets to have this shifted, embedded cursor. Now we want to use these characters in a much more subtle way.

Basically, the Utilities program we referred to earlier accomplishes the cursor insertion in the following way. Extra versions (left and right shifted) of the current character set are stored away. When the cursor is to be inserted between two characters, say "A" and "B", the program defines two special characters (we'll use ASCII 0 and 1) as shifted versions of "A" and "B" and prints them where the "A" and "B" were. The redefinition of the characters is done through a "Control" call to the Console Driver, call 17 to be exact. This is the "partial character set download", which can load a maximum of eight character definitions. More on how this works can be found in your Standard Device Drivers Manual, and a reference back to a previous article (18) in this series, which used "call 17" extensively to do graphics on the text screen.

The reason for redefining the special characters instead of "A" and "B" directly is that we don't want to affect any other occurrences of "A" and "B" which may be on the screen. As soon as the cursor moves on to some other characters, we print the old characters back where they were, and redefine the special characters according to the characters in the new location.

The Program

Our program starts with some of the same definitions, and adds some new ones:

```
 10   DIM
highr%(15),lowr%(15),carryr%(15),highl%(15),lowl%(15),carryl%(15)
 15   DIM lchar$(127),rchar$(127),flname$(9),vert%(9),horz%(9)
 20   DIM sleft%(255),sright%(255),char%(511),fstart%(9),fend%(9)
 30   INVOKE"/basic/request.inv","/basic/download.inv"
 40   GOSUB 4000
 45   GOSUB 6000
 60   name$=".console"
 75   PERFORM loadfont(@charset$)
```

Notice that the 'charl%' and 'charr%' arrays are replaced in this version with string arrays 'lchar$' and 'rchar$'. Each occurance of the array contains a string defining an individual ASCII character. This was done because the "Control" invokable requires a string as a parameter for the call. Other new arrays are 'flname$','vert%','horz%', 'fstart%' and 'fend%'. These arrays define the fields we will use in mocking up a simple data entry screen as an example of how to use the field editing commands.

Next comes the initalization routine at line 4000. You should copy lines 4000 through 4320 from the previous program, since nothing has changed in that part. That goes for the little entry routine in line 5000 as well. The changed part of the initialization is below, with the major modification designed to create the string arrays 'lchar$' and 'rchar$':

```
4400   PRINT:PRINT"Preparing the character fonts"
4410   FOR i=0 TO 511 STEP 4
4415     k=i/4:lchar$(k)="":rchar$(k)=""
4420     FOR j=0 TO 2:a$=HEX$(char%(i+j))
4430       l=TEN(MID$(a$,1,2)):r=TEN(MID$(a$,3,2))
4440       lchar$(k)=lchar$(k)+CHR$(sleft%(l))+CHR$(sleft%(r))
4450       rchar$(k)=rchar$(k)+CHR$(sright%(l))+CHR$(sright%(r))
4460       NEXT j
4470     a$=HEX$(char%(i+3)):l=TEN(MID$(a$,1,2))
4480     lchar$(k)=lchar$(k)+CHR$(sleft%(l))+CHR$(127)
4490     rchar$(k)=rchar$(k)+CHR$(sright%(l))+CHR$(254)
4500     NEXT i
4510   RETURN
```

Each element of the string arrays now contains the eight bytes required to define that particular character, either right or left shifted. Next we have an initialization routine to set the field names and parameters described earlier:

```
6000   DATA 5
6005   DATA "Name: ",6,1,7,30
6010   DATA "Address: ",8,1,10,40
6015   DATA "City: ",10,1,7,26
6020   DATA "State: ",12,1,8,9
6025   DATA "Zip: ",14,1,6,10
6050   READ n:max.field=n-1
6055   FOR i=0 TO max.field
6060     READ flname$(i),vert%(i),horz%(i),fstart%(i),fend%(i)
6065     NEXT i
6090   lcursor$=CHR$(128):rcursor$=CHR$(129):cursor$=lcursor$+rcursor$
6095   blank$="                                              "
6100   RETURN
```

Notice that in line 6090 above that definitions are established for versions of the cursor, composed of the characters 0 and 1 (using the values 128 and 129 makes them printable). The single character cursor definitions are for opposite ends of an individual field, and the 'cursor$' definition is for insertions in the middle of a field.

Now that all the definitions are established, it's time to put fields on the screen and start editing:

```
500   HOME
505   FOR field=0 TO max.field
510     VPOS=vert%(field):HPOS=horz%(field):PRINT flname$(field);
520     flen=fend%(field)-fstart%(field)+1
530     cpos=1
540     value$=MID$(blank$,1,flen)
```

The lines above set up a loop to process all the fields, and then position the cursor, print the name of the field, set 'cpos' (the current position within the field) and clear the field value 'value$' to blanks.

```
550     HPOS=fstart%(field):PRINT value$;:HPOS=fstart%(field)+cpos-1
560     IF cpos>1 THEN 590
570     rval%=ASC(MID$(value$,1,1))
575     ctrlist$=CHR$(1)+CHR$(1)+lchar$(rval%)
580     PERFORM control(%17,@ctrlist$)name$
585     PRINT rcursor$;:GOTO 650

590     lval%=ASC(MID$(value$,cpos-1,1)):rval%=ASC(MID$(value$,cpos,1))
600     ctrlist$=CHR$(2)+CHR$(0)+rchar$(lval%)+CHR$(1)+lchar$(rval%)
610     PERFORM control(%17,@ctrlist$)name$
620     HPOS= HPOS-1:PRINT cursor$;
```

The routine from 570 to 585 handles the case of the current position being the extreme lefthand position in the field. 'Rval%' is set to the first character in 'value$' and a control list is built in line 575 with the shifted definition of that character as ASCII 1. The control call in 580 redefines ASCII 1, and then 585 prints it to the screen. Line 590-620 handles the case of mid-string positions, and redefines the characters on both sides of the cursor position.

```
650     GET a$:a=ASC(a$)
660     IF a<32 OR a>127 THEN 800
670     IF cpos=flen THEN 750:ELSE:IF cpos>flen THEN 650
675     SUB$(value$,cpos+1)=MID$(value$,cpos,flen-cpos)
680     SUB$(value$,cpos)=a$
690     cpos=cpos+1
700     GOTO 550
```

Lines 650-700 accept input from the user, checking it for control characters and "open-apple" commands. If it is an ordinary character, it is inserted into 'value$' and the routine jumps back to 550 to display the new version of the string. Notice the check made in line 670 to see if the cursor is at the right-hand end of the field. That situation is processed in line 750:

```
750     SUB$(value$,cpos)=a$:IF cpos=1 THEN 760
```

```
 755     HPOS= HPOS-2:PRINT MID$(value$,cpos-1,1);
 760     lval%=a:ctrlist$=CHR$(1)+CHR$(0)+rchar$(lval%)
 770     PERFORM control(%17,@ctrlist$)name$
 780     HPOS=fend%(field):PRINT lcursor$;:cpos=cpos+1:GOTO 650
```

The routine above is the "flip side" of the routine at 560, and sets up a single character definition at the right edge of the field.

The routines above handled simple character inserts. Now comes the control character processing for all the fun stuff:

```
 800     IF a>127 THEN a=a-128:GOTO 900
 805     IF a=9 THEN 970
 810     IF a<>8 THEN 830
 815     IF cpos=1 THEN 650:ELSE IF cpos<flen+1 THEN 825
 817     HPOS= HPOS-1:PRINT MID$(value$,cpos-1,1);
 820     HPOS= HPOS-1:cpos=cpos-1:GOTO 560
 825     HPOS= HPOS-2:PRINT MID$(value$,cpos-1,2);
 827     cpos=cpos-1:HPOS= HPOS-2:GOTO 560
```

Line 800 dispatches the use of "open-apple" keys to line 900, and 805 sends the routine to the next field if "TAB" is pressed. Next comes the routine for the cursor backarrow, ASCII 8. This must first restore the character on the right of the cursor by printing it from 'value$', and then jump back to 560 to do the new cursor display.

```
 830     IF a<>21 THEN 860
 835     IF cpos>flen THEN 650:ELSE
         IF cpos=flen THEN a=ASC(MID$(value$,cpos,1)):GOTO 755
 840     IF cpos=1 THEN HPOS= HPOS-1:PRINT MID$(value$,cpos,1);:GOTO 850
 845     HPOS= HPOS-2:PRINT MID$(value$,cpos-1,1);:HPOS= HPOS+1
 850     cpos=cpos+1:GOTO 560
```

The routine above does the same thing for ASCII 21 (forward arrow), and returns to either 755 if at the end of the field or 560 if in the middle.

```
 860     IF a=13 THEN SUB$(value$,cpos,flen-cpos+1)=blank$:cpos=1:GOTO
550
 870     IF a=27 THEN 990:ELSE:GOTO 650
```

Line 860 above handles the carriage return by chopping off anything to the right of the cursor and sending the cursor back to the beginning of the field, exactly as it would be done on a typewriter. Line 870 handles ESCAPE, by exiting to the wrapup routine in line 990.

```
 900     IF a<>8 THEN 920
 905     IF cpos=1 THEN 650
 910     SUB$(value$,cpos-1)=MID$(value$,cpos)+" "
 915     cpos=cpos-1:GOTO 550
```

```
920     IF a<>21 THEN 960
925     IF cpos>flen THEN 650
930     SUB$(value$,cpos)=MID$(value$,cpos+1)+" "
935     GOTO 550
```

Remember that lines 900-935 above can only be reached if the "open-apple" key was pressed along with another key. In this case, 900-915 handles "open-apple <--", which deletes characters from the cursor position back to the beginning of the field, one character at a time. Lines 920-935 handle the opposite, "open-apple -->" which deletes characters in from the cursor position to the end of the field, also one at a time. In this way all the functions of the System Utilities editing are duplicated.

```
960     GOTO 650
970     HPOS=fstart%(field):PRINT value$;
980     result$(field)=value$
985     NEXT field
990   PRINT:PRINT:FOR i=0 TO max.field:PRINT result$(i):NEXT
```

Lines 970-985 wrap up the processing of a field by reprinting it to insure correctness and storing the value in the 'result$' array for future use. Then 985 takes the program back to process the next field, or on to 990 to print out the accumulated data.

Into the Home Stretch

One last bit of wrapup and we're finished. Since the program could use any font, we restore the standard font at the end, cleanup the invokables, and end, like so:

```
2000   stdset$=CHR$(34)+"/basic/standard"+CHR$(34)
2005   PERFORM getfont(@stdset$,@charset$)
2010   PERFORM loadfont(@charset$)
2020   INVOKE
2030   END
```

There you have it. This is no great shakes as a data entry program, of course. Its purpose was to give the editing posibilities of the new cursor insert mode a workout. With some spiffing up you could use this as a decent routine, however, especially if the performance was improved. No real effort was made to streamline the typing of characters, so since it has a lot of work to do, you can easily get ahead if you're a fast typist.

Another area for improvement is to store the string arrays which define the characters in disk files and read them in at run time. This should take less time than generating the characters each time.

Finally, there are some differences in how this program works and the editing capability of the System Utilities program. One nice thing that utilities does is to flash the underline part of the cursor. This is accomplished by setting the high-order bit in the bottom row of the character definitions, and turning on inverse mode before printing the cursor. Since this also requires that you create the INVERSE version of the shifted character (so the inverse of inverse is normal, get it?), that got a little messy for this article. Those of you with patience will have it rewarded by being able to exactly duplicate the utilities program. Well, now that that's said, there is one "little" thing more. Utilities also handles the problem of characters which have descenders. Our program ignores descenders on characters and just puts the underline in regardless. Utilities actually moves the character up one row, which it can do since only lower case letters have descenders. Some people find this bothersome, but it eliminates the legibility problem which sometimes occurs when the descender is cut off.

The suggestions in the paragraph above come under the heading of SMOP's. SMOP stands for "Simple Matter Of Programming" which is roughly equivalent to "it can easily be shown that..." for mathematicians. There is no task which you can't accomplish with a computer, it's just a SMOP.

Until next time, may all you SMOP's be little ones!

Exploring Business Basic - Part XXIII

Last month we undertook a challenge to use the flexibility of the "soft" (programmable) character set of the Apple III to do something usually reserved for systems using bit-mapped graphics displays, that is, to create a cursor on the screen which actually lived between two adjacent screen characters. This was done in response to the need to know exactly where inserting of typed characters would occur. The example was borrowed from the Apple III System Utilities program, where this "insert mode" cursor is used in the input fields. Some other Pascal programs use this technique, but it turned out that it was fairly easy to implement in Basic as well ('Yay, Basic!). However, the way things were done last time was somewhat different from the Utilities implementation, and so the article concluded with some challenges to make the routine better and more useful in data entry programs, as well as more like the way things work in Utilities. Shame on this column! This month's intrepid episode revisits the program and the previous column's parting challenges and delivers a more robust and useful version. For those of you who undertook to solve the problems raised last time, this will serve as a check on one way (not necessarily the best) to attack the solution.

Quick as a Wink

Just to refresh your memory, the way we attacked to problem of putting a cursor between two characters was to create two new sets of character set definitions in which each character was moved to the left or right one pixel position within the character cell (since most characters occupy 5X7 dots within a 7X8 cell). A vertical row of dots was put into the vacant space left when the shift occured. By redefining two adjacent characters on the screen with a left-shifted and a right-shifted version, the result was two standard characters with a double width bar between them, indicating where the insertion would occur. To make things clearer, the bottom row of pixels was turned on as well, in the space normally occupied by the descenders of lower case characters.

While this was a good first pass, the System Utilities program does several more things which make this easier to use and more legible. First, to draw attention to where the cursor is, the program flashes the underline below the two characters, instead of leaving it constantly on. Secondly, the program handles the problem of the flashing underline destroying lower case descenders by physically moving the character up one row so that the descenders are preserved. It can do this because only five lower case characters have descenders (g,j,p,q,y) and in each case the character can be moved up without losing its appearance. Granted, the "j" loses its top dot, but life is like that.

In the program below, we'll first tackle the problem of making the underline "wink" (rhymes with blink) at us, and then look at a method of shifting up the five lower case characters. To get the blinking underline, it's necessary to remember that the high order bit in a character definition (the left-most or Most Significant Bit, some people would say) is used to control blinking for that line of pixels in the character cell, but only if the character is written in "inverse" mode. This means that you can have normal mode and either inverse or blinking, but not both.

Programming Inverse

The information above suggests a simple solution, namely to set the high order bit of the last row of the cursor character definitions and print them in inverse mode. However, since we must print the whole character in inverse, for the top seven rows to come out normally, it is necessary to create an inverse (black on white) character definition, which, when printed in inverse will result in the "normal" white pixels on black. Fortunately, as we will see, inverting the character definition is much simpler than shifting it. Those of you with exceptional memories (or exceptional faithfullness to this column) will remember that this technique was covered when we created the character and shape editor in the February episode. Then, as now, the way to invert a bit pattern is to subtract it from a bit pattern of all "ones". Thus:

```
 1111111          1111111
-1001101         -0011101
--------         --------
 0110010          1100010          ,etc.
```

In decimal, it's as simple as subtracting from 127, 255 or any other number which is all "ones" in binary.

Dividing up the Work

One of the other challenges from last time was to divide the program into two parts, one which created the shifted (and now inverted) characters and the field definitions, and the other of which used the character definitions to do data entry into the fields. The major reason for this division of labor was to avoid spending the time inverting and shifting the characters each time the data entry program was run. Also, by building files of shifted character definitions, field names and specifications, it would be possible to use the same data entry program to enter many different sets of data, depending on which definition file the data entry program used. This is much more the way that real applications are built.

Since the details of the character shifting process were fully described last time, the description of the program below will just note changes from the previous version:

```
 10   DIM
highr%(15),lowr%(15),carryr%(15),highl%(15),lowl%(15),carryl%(15)
 15   DIM lchar$(127),rchar$(127),sleft%(255),sright%(255),char%(511)
 20   DIM flname$(20),vert%(20),horz%(20),fstart%(20),fend%(20)
 30   INVOKE"/basic/request.inv","/basic/download.inv"

 35   PRINT"Initializing values, please wait"
 40   GOSUB 4000
 45   GOSUB 6000
```

The section above uses the same routines at lines 4000 and 6000 to build the character definitions and field descriptions, but this time, instead of using these definitions immediately, the program section below writes this information out to a definition storage file:

```
 50   INPUT"Storage file: ";a$
 60   IF a$="" THEN 500
 70   OPEN#1,a$
 80   PERFORM filwrite(%1,@charset$,%1024)
 90   WRITE#1,10;font$
 100   FOR i=0 TO 127
 110     WRITE#1;lchar$(i),rchar$(i)
 120     NEXT i
 130   WRITE#1,20;max.field+1
 140   FOR i=0 TO max.field
 150     WRITE#1;flname$(i),vert%(i),horz%(i),fstart%(i),fend%(i)
 160     NEXT i
 170   PRINT"Information stored."
```

Line 80 writes out the character set which corresponds to the shifted definitions in "lchar$" and "rchar$". This insures that the data entry program will use consistant definitions for regular characters and redefined ones, no matter what font is selected in this program. After writing out the character set, the shifted definitions are written in lines 90 through 120. Note that the pathname of the original font is written for reference. In 130-160 then field name definitions are written, along with the display location and length information. At that point this program is finished:

```
 500   REM end of program
 510   CLOSE:INVOKE
 520   END
```

Most of the routine at line 4000 which builds the shifted character definitions is the same, but is repeated so that you can recreate the whole program:

```
 4000   DATA 0,2,4,6,0,2,4,6,8,10,12,14,8,10,12,14
 4010   DATA 1,3,5,7,9,11,13,15,1,3,5,7,9,11,13,15
 4020   DATA 0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1
 4030   DATA 4,4,5,5,6,6,7,7,12,12,13,13,14,14,15,15
 4040   DATA 0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7
 4050   DATA 0,8,0,8,0,8,0,8,0,8,0,8,0,8,0,8

 4100   FOR i=0 TO 15:READ highl%(i):NEXT:FOR i=0 TO 15:READ lowl%
(i):NEXT
 4110   FOR i=0 TO 15:READ carryl%(i):NEXT
 4120   FOR i=0 TO 15:READ highr%(i):NEXT:FOR i=0 TO 15:READ lowr%
(i):NEXT
 4130   FOR i=0 TO 15:READ carryr%(i):NEXT
 4150   FOR i=0 TO 15:FOR j=0 TO 15
 4160       sleft%(i*16+j)=16*(highl%(i)+carryl%(j))+lowl%(j)
 4170       sright%(i*16+j)=16*highr%(i)+lowr%(j)+carryr%(i)
 4180       NEXT:NEXT

 4200   prompt$="Character font pathname: ":GOSUB 5000
 4210   IF error THEN RETURN
 4220   ON ERR GOTO 4260
 4230   font$=CHR$(34)+a$+CHR$(34):charset$="char%"
 4240   PERFORM getfont(@font$,@charset$)
 4250   OFF ERR:PRINT"Font loaded":GOTO 4400
 4260   ON ERR GOTO 4300
 4270   OPEN#1,a$:PERFORM filread(%1,@charset$,%1024,@ret%)
 4280   IF ret%=1024 THEN OFF ERR:GOTO 4400
 4300   OFF ERR:PRINT a$" is not a valid character font file"
 4310   IF TYP(1)=0 THEN CLOSE#1:DELETE a$:GOTO 4200
 4320   CLOSE#1:GOTO 4200
```

The shift preparation routines below contain the changes for inverting the character set to allow for our blinking underline cursor:

```
 4400   PRINT:PRINT"Preparing the character fonts"
 4410   FOR i=0 TO 511 STEP 4
 4415     k=i/4:lchar$(k)="":rchar$(k)=""
 4420     FOR j=0 TO 2:a$=HEX$(char%(i+j))
 4430       l=TEN(MID$(a$,1,2)):r=TEN(MID$(a$,3,2))
 4440       lchar$(k)=lchar$(k)+CHR$(127-sleft%(l))+CHR$(127-sleft%(r))
 4450       rchar$(k)=rchar$(k)+CHR$(127-sright%(l))+CHR$(127-sright%
(r))
 4460       NEXT j
```

```
4470     a$=HEX$(char%(i+3)):l=TEN(MID$(a$,1,2)):r=TEN(MID$(a$,3,2))
4480     lchar$(k)=lchar$(k)+CHR$(127-sleft%(l))+CHR$(256-sleft%(r))
4490     rchar$(k)=rchar$(k)+CHR$(127-sright%(l))+CHR$(319-sright%(r))
4500     NEXT i
4510   RETURN
```

Notice that in lines 4440,4450,4480 and 4490 that the value of the individual row definition is first subtracted from 127 to form the inverse of the regular value. We use 127 (seven bits of ones) because we want the high order (flash) bit to be left off on all rows but the bottom. The bottom row left is handled in line 4480, where subtracting from 256 not only inverts the bottom seven bits (the character's pixel definition) and inverts (turns on) the flash bit, but also (255+1) turns on the low order bit, to be consistant with the state of the other bits (remember that sleft% forces the low order bit to one for the cursor center line, and the invert operation performed by the subtraction turns it back off). Line 4490 performs a similar operation, except this time we must subtract from 319 (255+64) which has the effect of inverting all the bits and then turning the bit next to the flash (high order) bit on. Trust it, it works!

The rest of the routine is very similar to the version from last time:

```
5000   PRINT prompt$;:INPUT"";a$
5010   IF a$="" THEN error=1:RETURN
5020   error=0:RETURN

6000   DATA 5
6005   DATA "Name: ",6,1,7,30
6010   DATA "Address: ",8,1,10,40
6015   DATA "City: ",10,1,7,26
6020   DATA "State: ",12,1,8,9
6025   DATA "Zip: ",14,1,6,10
6050   READ n:max.field=n-1
6055   FOR i=0 TO max.field
6060     READ flname$(i),vert%(i),horz%(i),fstart%(i),fend%(i)
6065     NEXT i
6100   RETURN
```

Descending Ever Upward

As was mentioned before, it is possible to modify the character shifting and inversion routine to more nearly match how the System Utilities program behaves. The lines below, when added to the program above, will move the definitions of the characters g,j,p,q,y up one row of pixels, so that a blinking underline can be used without distorting the character appearance. This has the disadvantage of making the character appear to bounce up when the cursor moves over it, but its purely a matter of personal taste. The best way is to make

the modifications and try creating both kinds of definitions. Use the one you like and which makes the most sense to you. Anyway, here are the changes:

```
 4432     IF k<103 OR j>0 THEN 4440
 4433     IF k<>103 AND k<>106 AND k<>112 AND k<>113 AND k<>121 THEN
4440
 4435     lchar$(k)=CHR$(127-sleft%(r)):rchar$(k)=CHR$(127-sright%(r)):
          GOTO 4460

 4472     IF k<103 THEN 4480
 4473     IF k<>103 AND k<>106 AND k<>112 AND k<>113 AND k<>121 THEN
4480
 4475     lchar$(k)=lchar$(k)+CHR$(127-sleft%(l))+CHR$(127-sleft%(r))+
          CHR$(255)
 4478     rchar$(k)=rchar$(k)+CHR$(127-sright%(l))+CHR$(127-sright%(r))+
          CHR$(255)
 4479     GOTO 4500
```

As you can see, it works by skipping the loading of the top row (line 4435) and adds an extra row (now the new bottom row) which just consists of a solid flashing line (the chr$(255) in line 4475 and 4478).

Spreading the Word

Having created and stored the character definitions and the data entry definitions, it's time to show the data entry program which takes advantage of all this work. Because the program below follows the program from last time almost exactly in the way it displays the fields and accepts data, only the changed parts will be described in detail:

```
 10   DIM
highr%(15),lowr%(15),carryr%(15),highl%(15),lowl%(15),carryl%(15)
 15   DIM lchar$(127),rchar$(127),flname$(9),vert%(9),horz%(9)
 20   DIM sleft%(255),sright%(255),char%(511),fstart%(9),fend%(9)
 30   INVOKE"/basic/request.inv","/basic/download.inv"

 40   GOSUB 3000
 50   IF error THEN 2000
 60   name$=".console"
 70   blank$="
"
 75   lcursor$=CHR$(128):rcursor$=CHR$(129):cursor$=lcursor$+rcursor$
 80   INPUT"Name of recording file: ";a$
 85   IF a$="" THEN a$=".console"
 90   OPEN#2,a$
```

Our first task after allocating the arrays is to read in the definitions from the file created in the previous program. That's done at line 3000 and looks like this:

```
3000   INPUT"Screen definition file: ";a$
3005   error=0:IF a$="" THEN error=1:RETURN
3010   charset$="char%"
3015   ON ERR GOTO 3900
3020   OPEN#1,a$
3025   PERFORM filread(%1,@charset$,%1024,@ret%)
3030   OFF ERR
3035   IF ret%<>1024 THEN 3900
3040   PERFORM loadfont(@charset$)
3100   READ#1,10;font$
3105   FOR i=0 TO 127
3110     READ#1;lchar$(i),rchar$(i)
3115     NEXT i
3120   READ#1,20;n:max.field=n-1
3125   FOR i=0 TO max.field
3130     READ#1;flname$(i),vert%(i),horz%(i),fstart%(i),fend%(i)
3135     NEXT i
3200   PRINT"Information loaded"
3205   RETURN

3900   PRINT"Invalid definition file, try again"
3905   OFF ERR
3910   GOTO 3000
```

The routine above reverses the process used to originally write the file, and stores the information used by the data entry routine in the appropriate arrays.

Glancing back up to the rest of the routine in lines 80-90, we now allow a file to be opened to store the data which will be input. The example used here is a simple one, and typically would have lots more complications, like creating keys and writing out data into a database structure, etc. but for now the example given simplifies the issues. Notice that the default is the screen (.console) which gives a convenient way to test that the data we see is really the data which was accepted as input.

Next we come to the input routine itself, which was heavily described last time. It basically cycles through all the fields and accepts data while displaying the "insert mode" cursor created by the new character definitions.

```
500   HOME
505   FOR field=0 TO max.field
510     VPOS=vert%(field):HPOS=horz%(field):PRINT flname$(field);
520     flen=fend%(field)-fstart%(field)+1
530     cpos=1
```

```
 540     value$=MID$(blank$,1,flen)
 550     HPOS=fstart%(field):PRINT value$;:HPOS=fstart%(field)+cpos-1
 560     IF cpos>1 THEN 590
 570     rval%=ASC(MID$(value$,1,1))
 575     ctrlist$=CHR$(1)+CHR$(1)+lchar$(rval%)
 580     PERFORM control(%17,@ctrlist$)name$
 585     INVERSE:PRINT rcursor$;:NORMAL:GOTO 650

 590     lval%=ASC(MID$(value$,cpos-1,1)):rval%=ASC(MID$(value$,cpos,1))
 600     ctrlist$=CHR$(2)+CHR$(0)+rchar$(lval%)+CHR$(1)+lchar$(rval%)
 610     PERFORM control(%17,@ctrlist$)name$
 620     HPOS= HPOS-1:INVERSE:PRINT cursor$;:NORMAL

 650     GET a$:a=ASC(a$)
 660     IF a<32 OR a>127 THEN 800
 670     IF cpos=flen THEN 750:ELSE:IF cpos>flen THEN 650
 675     SUB$(value$,cpos+1)=MID$(value$,cpos,flen-cpos)
 680     SUB$(value$,cpos)=a$
 690     cpos=cpos+1
 700     GOTO 550

 750     SUB$(value$,cpos)=a$:IF cpos=1 THEN 760
 755     HPOS= HPOS-2:PRINT MID$(value$,cpos-1,1);
 760     lval%=a:ctrlist$=CHR$(1)+CHR$(0)+rchar$(lval%)
 770     PERFORM control(%17,@ctrlist$)name$
 780     HPOS=fend%(field):INVERSE:PRINT
lcursor$;:NORMAL:cpos=cpos+1:GOTO 650

 800     IF a>127 THEN a=a-128:GOTO 900
 805     IF a=9 THEN 970
 810     IF a<>8 THEN 830
 815     IF cpos=1 THEN 650:ELSE IF cpos<flen+1 THEN 825
 817     HPOS= HPOS-1:PRINT MID$(value$,cpos-1,1);
 820     HPOS= HPOS-1:cpos=cpos-1:GOTO 560
 825     HPOS= HPOS-2:PRINT MID$(value$,cpos-1,2);
 827     cpos=cpos-1:HPOS= HPOS-2:GOTO 560
 830     IF a<>21 THEN 860
 835     IF cpos>flen THEN 650:ELSE IF cpos=flen THEN
         a=ASC(MID$(value$,cpos,1)):GOTO 755
 840     IF cpos=1 THEN HPOS= HPOS-1:PRINT MID$(value$,cpos,1);:GOTO 850
 845     HPOS= HPOS-2:PRINT MID$(value$,cpos-1,1);:HPOS= HPOS+1
 850     cpos=cpos+1:GOTO 560
 860     IF a=13 THEN SUB$(value$,cpos,flen-cpos+1)=blank$:cpos=1:GOTO
550
```

```
870     IF a=27 THEN 990
875     GOTO 650

899     REM routine below handles "open-apple" keys
900     IF a<>8 THEN 920
905     IF cpos=1 THEN 650
910     SUB$(value$,cpos-1)=MID$(value$,cpos)+" "
915     cpos=cpos-1:GOTO 550
920     IF a<>21 THEN 650
925     IF cpos>flen THEN 650
930     SUB$(value$,cpos)=MID$(value$,cpos+1)+" "
935     GOTO 550

969     REM put value into the result array and get next value
970     HPOS=fstart%(field):PRINT value$;
980     result$(field)=value$
985     NEXT field
```

Notice above that in lines 585, 620 and 780 we now turn on inverse mode, print the cursor characters, and then turn normal back on. This, in combination with the high bit "on" in the character's bottom row, creates the flashing underline we want. After all the fields are filled (remember that TAB gets us to the next field and ESCAPE jumps out of input mode), we write out the results in the storage file:

```
990   REM end of input cycle, write out results
1000   PRINT:PRINT
1005   FOR i=0 TO max.field
1010     PRINT#2;result$(i)
1015     NEXT i
1020   PRINT"Record written"
1030   INPUT"Continue? ";a$
1035   IF a$="" THEN 500
1040   a$=MID$(a$,1,1):IF INSTR("Yy",a$) THEN 500
```

As was said previously, the routine from lines 1005 to 1015 can, and probably needs to be much more complicated to be useful. After prompting to see if there are more records to be entered, line 1040 either takes the user back for more input, or terminates using the routine below:

```
2000   stdset$=CHR$(34)+"/basic/standard"+CHR$(34)
2005   PERFORM getfont(@stdset$,@charset$)
2010   PERFORM loadfont(@charset$)
2020   CLOSE:INVOKE
2030   END
```

Lines 2000-2030 load the standard character set back using "getfont" and "loadfont" and terminate the program.

Exiting Data Entry

The data entry program above with its nice field handling capabilities can be the basis for lots of applications. Because the definition of the data to be entered is stored separately from the data entry program itself, one program can truly serve many needs. This is much more characteristic of the way real database query and update programs work, with one piece of program code operating on lots of different database definitions. One thing which would make this combinition much more useful is if you modified the definition program to allow the input and editing of the field definitions, rather than having to type them into data statements. You might even want to store the field definitions in separate files, since one set of character definitions would serve many different sets of fields, and would save considerable disk space as well.

Tune in next time for a special treat, as this intrepid column brings you some news which will delight every Apple III owner (and make some of you Apple // owners a bit jealous!). 'Till then...

Appendix -- Additional Information

P O W E R C A T III

Sorted Catalog List for Device : {MIXED}

Catalogs Referenced in this Listing :

```
Ref Nmbr          Catalog Name  {id info}

1               /THREE.SIG.1017A/ARTICLE1/
2               /THREE.SIG.1017A/ARTICLE2/
3               /THREE.SIG.1017B/ARTICLE3/
4               /THREE.SIG.1017B/ARTICLE4/
5               /THREE.SIG.1017B/ARTICLE5/
6               /THREE.SIG.1018A/ARTICLE6/
7               /THREE.SIG.1018A/ARTICLE7/
8               /THREE.SIG.1018B/ARTICLE8/
9               /THREE.SIG.1018B/ARTICLE9/
10              /THREE.SIG.1019A/ARTICLE10/
11              /THREE.SIG.1019A/ARTICLE11/
12              /THREE.SIG.1019A/ARTICLE12/
13              /THREE.SIG.1019B/ARTICLE12/
14              /THREE.SIG.1019B/ARTICLE13/
15              /THREE.SIG.1019B/ARTICLE14/
16              /THREE.SIG.1019B/ARTICLE15/
17              /THREE.SIG.1020A/ARTICLE16/
18              /THREE.SIG.1020A/ARTICLE17/
19              /THREE.SIG.1020A/ARTICLE18/
20              /THREE.SIG.1019B/ARTICLE18/
21              /THREE.SIG.1019B/ARTICLE19/
22              /THREE.SIG.1021A/ARTICLE20/
23              /THREE.SIG.1021A/ARTICLE21/
24              /THREE.SIG.1021B/ARTICLE21/
25              /THREE.SIG.1021B/ARTICLE22/
26              /THREE.SIG.1021B/ARTICLE23/
```

```
File  Name      Ref#  Type   Blks

ARC                7 BASIC  00005
ARC.SUB            7 BASIC  00003
ARC.TIME           7 BASIC  00006
ARTICLE.1          1 TEXT   00040
ARTICLE.2          2 TEXT   00039
ARTICLE.3          3 TEXT   00044
ARTICLE.4          4 TEXT   00055
ARTICLE.5          5 TEXT   00052
ARTICLE.6          6 TEXT   00060
ARTICLE.7          7 TEXT   00052
ARTICLE.8          8 TEXT   00043
ARTICLE.9          9 TEXT   00067
ARTICLE.10        10 TEXT   00068
ARTICLE.11        11 TEXT   00064
ARTICLE.12        13 TEXT   00033
ARTICLE.13        14 TEXT   00023
ARTICLE.14        15 TEXT   00054
ARTICLE.15        16 TEXT   00054
ARTICLE.16        17 TEXT   00057
ARTICLE.17        18 TEXT   00070
ARTICLE.18        20 TEXT   00064
ARTICLE.19        21 TEXT   00069
ARTICLE.20        22 TEXT   00078
ARTICLE.21        23 TEXT   00074
ARTICLE.22        25 TEXT   00049
ARTICLE.23        26 TEXT   00040
B.SORT.2D         16 BASIC  00001
BINARY.SORT       16 BASIC  00005
BINARY.SORT       17 BASIC  00005
BOUNCE.PRIOR      22 BASIC  00005
BOUNCE.SCRUB      22 BASIC  00004
BUBBLE.SORT       15 BASIC  00001
BUBBLE.SORT       16 BASIC  00001
BUBL.PNTR.SORT    15 BASIC  00001
BUBL.PNTR.SORT    16 BASIC  00001
BUG.BOX           22 BASIC  00005
```

```
File  Name      Ref#  Type   Blks

BUG.FONT          18  FONT   00003
BUG.FONT          19  FONT   00003
BUG.FONT          21  FONT   00003
BUG.FONT.1        18  FONT   00003
BUG.FONT.1        19  FONT   00003
BUG.MANIA         19  BASIC  00008
BUILD.FILE.1      26  BASIC  00006
BUILD.INV.BYTES    8  BASIC  00001
CHANGES.PROG       4  BASIC  00003
CHAR.ARROW        18  DATA   00007
CIRCLE             7  BASIC  00001
CIRCLE.ASPECT      7  BASIC  00001
CIRCLE.FINAL       7  BASIC  00003
CIRCLE.PROG        7  BASIC  00003
CIRCLE.SCALE       7  BASIC  00001
CIRCLE.SUB         7  BASIC  00001
COMMAND.TEST       8  BASIC  00008
CREATE.TRASH      10  BASIC  00003
CREATE.TRASH      11  BASIC  00003
CREATEJUNKFILE    16  BASIC  00001
DATA.PARTS.PROG    3  BASIC  00007
DATABASE.PROG      9  BASIC  00013
DB.BTREE          17  BASIC  00019
DECODE            10  BASIC  00003
DECODE.DUMP       10  BASIC  00004
DEMO.MOVE.FONT    25  BASIC  00005
DEMO.PRIORITY     19  BASIC  00005
DIR.DUMP.OUT       6  TEXT   00001
DIRECTORYDUMP      6  TEXT   00006
DUMP.DATA          5  TEXT   00005
DUMP.DIR           6  BASIC  00001
DUMP.OUT           5  TEXT   00013
DUMP.STATUS       10  BASIC  00004
DUMP.USING.GET     5  BASIC  00001
EDIT.FONT          8  BASIC  00010
EDIT.SHAPE        18  BASIC  00025
EDIT.SHAPE        21  BASIC  00025
EDITOR             8  BASIC  00009
ELLIPSE            7  BASIC  00001
EXTENDED.ADD       6  BASIC  00004
FAST.B.SORT       16  BASIC  00005
FAST.CIRCLE        7  BASIC  00001
```

```
FAST.MOVE          21  BASIC  00004
FILL.PIC            8  FOTO   00033
FINAL.B.SORT       16  BASIC  00005
FINAL.FIRE         22  BASIC  00006
FINAL.PROP.EDIT    24  BASIC  00011
FONTLOAD.SUB       19  BASIC  00001
FONTLOAD.SUB       21  BASIC  00001
FOR.NEXT.OUT        6  TEXT   00001
FORMAT.DUMP         5  BASIC  00001
FREE.FIRE          19  BASIC  00006
FREE.FIRE          21  BASIC  00006
GENERATE.JUNK       9 BASIC  00001
GET.STATUS         10 BASIC  00003
GOSSIPFILE         10 TEXT   00009
GOSSIPFILE         11 TEXT   00015
HASH.EXAMPLE        9 BASIC  00001
HASH.SUB            9 BASIC  00001
HASH.TEST           9 BASIC  00001
HEX.CONVERT         5 BASIC  00001
HIRES.CRAWL        19 BASIC  00001
HIRES.CRAWL        21 BASIC  00001
HIRES.DUMP          8 BASIC  00006
HSCROLL            10 BASIC  00001
HSCROLL.ARRAY      10 BASIC  00001
INPUT.DATA         26 BASIC  00007
INPUT.FIELDS       25 BASIC  00009
INPUTEXAMPLE        3 BASIC  00001
INSERT.MODE        25 BASIC  00005
INV.BYTES           8 DATA   00003
INVERT.FONTDEMO     8 BASIC  00003
INVERT.HEX          8 BASIC  00001
INVERT.INSERT      25 BASIC  00009
INVERT.INT          8 BASIC  00001
INVERT.SORT        15 BASIC  00005
INVERT.SORT        16 BASIC  00005
LAST.TRY           24 BASIC  00012
LIKE.UTILITIES     26 BASIC  00006
LINETO.CIRCLE       7 BASIC  00001
LIST.PARTS          3 BASIC  00001
LISTP              11 TEXT   00001
LISTROUTINE         4 TEXT   00001
LONG.ADD.SUB        6 BASIC  00001
LONG.INPUT.SUB      6 BASIC  00001
LONG.QUICK         15 BASIC  00004
```

```
LOOP.EXAMPLE      19 BASIC  00003
MODE.CIRCLE        7 BASIC  00003
MODE1.CIRCLE       7 BASIC  00001
MOVE.BUG          21 FONT   00003
MOVE.BY.THREE     21 BASIC  00003
MOVE.HEADS        21 BASIC  00004
MOVE.SINGLE       21 BASIC  00005
MOVE.SINGLE       22 BASIC  00005
MOVING.DOWNLOAD   25 BASIC  00006
MYPROGRAM          6 BASIC  00001
NEW.BUG.MANIA     19 BASIC  00008
NEW.CREATE.JUNK    5 BASIC  00001
NEW.FDEMO          8 BASIC  00003
NEW.INVERT        25 BASIC  00009
NEW.INVERT        26 BASIC  00009
NEW.MOVE.THREE    22 BASIC  00003
NEW.PROG.DATA      4 BASIC  00008
NEW.SCRUB         22 BASIC  00004
NEW.TRY           17 TEXT   00001
NEWCIRCLE          7 BASIC  00001
NEXT.SHAPE        21  DATA   00004
NOTE              12  TEXT   00001
NOWAIT.2BYTE      14  BASIC  00003
NUMBERFILE         3  TEXT   00003
NUMBERFILE1        3  DATA   00003
OUTPUT.CREATE      5  TEXT   00003
PARTS.LIST         4  BASIC  00001
PARTS.PROG         3  BASIC  00007
PARTS.PROG         5  BASIC  00009
PNTR.BUBBLE       15  BASIC  00003
PNTR.BUBBLE       16  BASIC  00003
PRIME              9  BASIC  00001
PRINT.DATA         3  BASIC  00001
PROGRAM1           1  BASIC  00001
PROGRAM1           2  BASIC  00001
PROGRAM2           1  BASIC  00001
PROGRAM2           2  BASIC  00001
PROGRAM3           2  BASIC  00001
PROGRAM4           2  BASIC  00001
PROGRAM5           2  BASIC  00001
PROGRAM6           2  BASIC  00003
PROP.PART         24  TEXT   00004
PROPORTION.EDIT   24  BASIC  00012
QUICK.BYTES        8  BASIC  00001
```

```
QUICK.FONT.DUMP   21  BASIC  00001
QUICKSORT         15  BASIC  00003
QUICKSORT         16  BASIC  00003
RANDOM.ARC         7  BASIC  00005
RANDOM.CIRCLE      7  BASIC  00004
RANDOM.TEST        9  BASIC  00001
RANDOM.TEXT        7  BASIC  00005
READ.BUBBLES      21  BASIC  00003
READ.SCRUB        22  BASIC  00004
READ.WAG          21  BASIC  00005
SCREEN.DATA.PGM   12  BASIC  00008
SCREEN.EDIT       11  BASIC  00008
SCREW.AROUND      22  BASIC  00003
SCROLL.4.WAYS     10  BASIC  00004
SCROLL.HIRES      19  BASIC  00001
SCROLL.HIRES      21  BASIC  00001
SCROLL.TEXT       19  BASIC  00001
SCROLL.VARIABLE   10  BASIC  00006
SCRUB.BUBBLES     21  BASIC  00003
SET.CONTROL       10  BASIC  00009
SHAPE.ARROW2      18  DATA   00004
SHAPE.LOAD.SUB    21  BASIC  00001
SHELL             15  BASIC  00003
SHELL             16  BASIC  00003
SHELL.SORT        15  BASIC  00001
SHELL.SORT        16  BASIC  00001
SHELL.SUB         15  BASIC  00001
SHORT.WAG         21  BASIC  00004
SIMPLE.BUBBLE     15  BASIC  00001
SIMPLE.HASH        9  BASIC  00001
SMOOTH.SCROLL     21  BASIC  00001
SORT.FRAME        15  BASIC  00001
SORT.FRAME        16  BASIC  00001
SORT.PROG.DATA     4  BASIC  00009
SORTSUB            4  TEXT   00003
STANDARD          19  FONT   00003
STANDARD.FONT     18  FONT   00003
STATUS.OUT        10  TEXT   00001
SWITCH            21  BASIC  00003
TABLE              7  BASIC  00001
TABLE             18  BASIC  00003
TABLE.CIRCLE       7  BASIC  00001
TEST.DUP           9  BASIC  00001
TEST.FIL.OUT       6  TEXT   00001
```

```
TEST.FILWRITE      6  BASIC  00003
TEST.FOR.NEXT      6  BASIC  00003
TEST.RECORD        6  BASIC  00001
TEST.REQUEST       6  DATA   00017
TEST.SHAPE        21  DATA   00004
TESTFORMAT         6  BASIC  00001
TESTNUMBER         3  BASIC  00001
TESTNUMBER1        3  BASIC  00001
TIME.CIRCLE        7  BASIC  00004
TIME.INVERT       15  BASIC  00005
TRY.STAT          10  BASIC  00003
TWO.SPD.SC2       10  BASIC  00005
TWO.SPD.SCROLL    10  BASIC  00005
TWO.SPD.SCROLL    11  BASIC  00005
ZOOM.SINGLE       22  BASIC  00005
```