Part I: Writing Transportable BASIC

Edward T. Ordman
Department of Mathematical Sciences
Memphis State University

If you think your programs might ever be used on another computer with a different dialect of BASIC – the suggestions in this two-part article can go a long way towards easing the transition. This month the author covers documentation, vocabulary, and readability. The article concludes next month with an overview of highly machine-sensitive issues such as input-output and graphics.

So you finally got your own computer. Unfortunately, it is not the same model you had at school. Or you've arrived at high school or college and the computer there is not the same one that your junior high school or high school had. What are you going to do with all the programs you have accumulated? My own school has just bought several of the new IBM Personal Computers – but most of the programs we have on hand were written for a mainframe or for our OSI microcomputers. Come to think of it, we are changing mainframes next semester, too!

Of course, all of these machines have a version of BASIC. (Some of them, in fact, have several versions of BASIC.) But, as is clear to anyone who has read a program written in Apple BASIC and wished he could run it on his Atari (or PET or TRS-80 or ...), all BASIC interpreters are not the same.

What is the solution? There is no ideal solution, for all cases. Some published programs are difficult to convert from one dialect to another. We can, however, in writing programs for ourselves, for friends, and perhaps even for publication, try to make our programs *transportable.* That is, we can write the programs so that they can be adapted to another machine with a minimum of difficulty.

Self-documenting

A program is easily transportable from one machine to another if it can be entered and run in the second machine with no substantial rewriting – certainly no changes in the underlying logic or algorithms – and a minimum of minor changes. The program should be self-explanatory so that it can be rewritten without knowledge of the original machine – a knowledge of the machine we are rewriting it for should be enough.

I have one fairly complex simulation program that was first written about 12 years ago for a PDP-8. It has since been rewritten, by me or by others, for S-100 bus machines in CBASIC, Apple, TRS-80, IBM Personal Computers and IBM 370's, Xerox Sigma 9, PDP-11, and enough other machines that I have lost count. I suspect that it would have been forgotten after the second or third transportation to a new machine, if it had not been written so that it was usually just a matter of typing it in again.

I should warn you at the outset that all this article considers is how to write the BASIC program. It does not address the problems of getting a program from one machine to another without having to key it in again. Increasingly, it is possible to connect the two computers over a phone line, directly or via one of the dial-up timesharing services, and move the program as a text file to avoid retyping. Nevertheless, the focus of this article is transportable *programming* techniques.

What can you do, when writing a program, to make it easily transportable? We will divide the strategy into five main parts: 1) minimal vocabulary; 2) in-program readability; 3) formal structuring; 4) careful attention to input-output; and 5) limited graphics.

Minimal Vocabulary

First, let's consider the question of vocabulary – what features of BASIC we should use. Apparently, whenever a company produces a new computer or a new version of BASIC, it feels compelled to add features not found in anyone else's BASIC. Often these features are convenient and may make programming for that machine easier. However, they make transporting a program much harder. If at all possible, such features should be avoided when writing with transportability in mind.

If we must use special features, they should be isolated in a subroutine near the end of the program and clearly labelled. The main program should stick to features found in virtually all versions of BASIC. This does not mean that string handling must be restricted to the limitations of Radio Shack Level 1 BASIC, which is an extreme example; nor are there universal rules as to what constructions are allowed. Some textbooks define "minimal BASIC" or restrict themselves in a similar way.

Educational institutions often belong to groups (consortia) which promote standards for exchanging programs; CONDUIT is one such educational group with a nice pamphlet on standard BASIC. If you have worked with several versions of BASIC, sticking to common features is a good guide for what will be transportable between them. For informal use, however, or for the individual who has just worked on one machine, here are the standards I have found useful in working with perhaps a dozen different machines, large and small.

Variables And Commands

Figure 1 suggests some guidelines for variable names, numbers, line numbers, DIM statements. Clearly, the list could be made much longer. For instance, how big can a real number be and not overflow? How small can a positive number be and still be distinguished from zero? Most BASIC programs do not depend critically on these figures, which may differ dramatically from one system to another.

Figure 1: Variables and Numbers

[illegible]

If your program does depend on them, you should probably make this explicit (and include a REMark giving the limits on your system). For instance, if your program has a variable X that gets closer and closer to zero as you go around a loop, and you exit the loop by testing IF X = 0 THEN ... , the program may behave very differently or even fail on another computer. Changing this to

```
500 IF ABS(X)<1E-50 THEN ... : REM USE A SMALL
    NON-ZERO NUMBER
```

will make the program transportable: the person converting it can check to see if the new computer will accept 1E-50. If it will not, he can substitute an acceptable number, e.g., 1E-30.

Figure 2: The most common statements

DATA	LET
DIM	NEXT
END	ON...GOTO
FOR...TO...STEP	PRINT
GOSUB	READ
GOTO	RESTORE
IF...THEN	RETURN
INPUT	

Figure 2 shows a limited list of BASIC commands – a very limited list. While almost every BASIC accepts more commands than these, they differ on which statements those are. For each command not on this list, there is some computer around that will not accept it. To make matters worse, computers differ substantially in how they interpret some of these commands. Some, for instance, do strange things on a STOP but allow END only as the last line of a program. The cure: place 9999 END as the last line of the program, and terminate anywhere else by GOTO 9999.

GOTO and GOSUB should be followed just by a line number. GOTO 500 is fine; avoid GOTO A even if your computer likes it. In the statement FOR X = A TO B STEP C, it is best to restrict A, B, and C to integers (or expressions evaluating to integers) and to avoid changing them inside the loop. NEXT must name just one variable for the corresponding FOR, e.g., NEXT X.

IF...THEN statements require special attention, since so many computers have so many different extensions. A few computers accept only statements such as IF Y> = Z THEN 830, prohibiting calculations, logical operations, and not allowing anything but a line number after THEN. I am not seriously suggesting that you keep things this simple: the extensions are extremely helpful. However, it *is* a good idea to keep things simple enough so that your statements can be translated into this form. This will be discussed further in the section on structure, next month.

Numeric And String Functions

Figure 3 shows the most commonly implemented numeric functions. Either most BASICs have these functions, or the programmer using the machine will be prepared to fake them somehow. Two deserve special mention: RND and TAB.

RND is implemented differently on almost every computer. Some use X = RND, some use

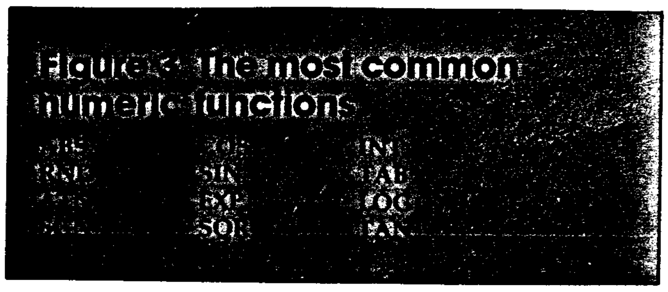


Figure 3: The most common numeric functions

ABS	COS	INT
RND	SIN	TAB
[illegible]	EXP	LOG
[illegible]	SQR	TAN

X = RND(0), some use RND(1), some use RANDOMIZE to start (seed) the random number generator and some do not. You should assume that every line containing RND will have to be rewritten. You should make this as easy as possible, by minimizing the number of lines involved and making your intention clear. If you need a random number in 20 different places in your program, do not have RND appear in 20 places; place it in a subroutine. That is, incorporate in your program

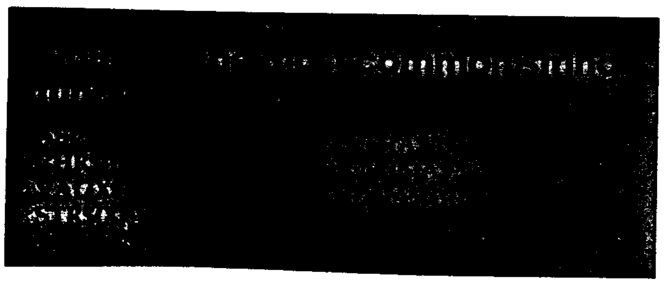
```
9000 REM *** GET RANDOM NUMBER, CHANG
     E FOR OTHER COMPUTERS ***
9010 X=RND(1):REM  RANDOM,0<X<1,NEW S
     EQUENCE EACH RUN
9020 RETURN
```

and then place GOSUB 9000 wherever needed in your program. Here is a more typical use, near the start of a game program:

```
150 N = INT(100*RND) + 1 :REM RANDOM INTEGER
    1 TO 100 ***************
```

Here the string of asterisks warns you, when transporting the program, that the line is likely to change. The remark tells what is wanted and will save a lot of time if the new computer achieves this by N = RND(100).

Turning briefly to TAB : there are computers that like TAB(N) (go to column N), those that like SPC(N) (print N spaces), those that like both, and those that like neither. Most people know how to juggle spacing on their own machine, so making your intention clear (by remarks or a sample printout) is probably more important than the exact way you write your PRINT statements. There will be more on this in the discussion of input-output, next month.


[illegible]

The functions given in Figure 4 are now remarkably widespread in *microcomputers*. It is probably safe to use all of them freely in that context. That is, if the person rewriting the program does not have LEFT$, he probably has a reasonbly direct substitute. You cannot count on the format produced by STR$ being the same from one machine to another – some pad with blanks on the left, some on the right, some not at all. Functions that match a substring are present on many machines, but absent on many others. Many systems will crash if you call LEFT$(A$,N) and A$ has less than N characters, so you should always test for this before you call LEFT$ even if your system does not insist on it.

Large computers differ substantially in how they handle strings, and are often *more* restrictive than small computers. ASC and CHR$ are frequently absent; many large computers do not even use the ASCII character set. Avoid extensive string manipulations, or at least place them in a subroutine, if your program may have to run on a large mainframe next year.

Readability

Next, if our program is to be readily transportable to another version of BASIC, it must be readable. First, can the reader understand our individual lines, and translate them for the new system? Second, can the reader understand our general strategy or procedure (our *algorithm*) well enough to debug the program if errors creep in, or if his BASIC interprets some command very differently than expected?

The most important consideration, for the second of these, is to make the program sufficiently modular and to provide appropriate REMarks for each module; this is addressed more in the discussion of structure, later. There are a number of "tricks of the trade" that make individual lines easier to read, however. Here are a few principles:

1. Leave plenty of space between line numbers. Even if you have only one command per line, some one-line commands on your system may become multiple commands on another. If you use several commands per line, the situation gets far worse. This is not to condemn all multiple-command-per-line statements, since they can add to the clarity of the program. Just remember that while your computer may allow:

```
500 INPUT "WHAT IS YOUR NAME?"; N$
```

someone else's may require

```
500 PRINT "WHAT IS YOUR NAME";
501 INPUT N$
```

This is an easy change if you left a line number available. It is quite possible for a complex one-line statement on one system to require six or eight lines on another.

2. Leave plenty of blanks in your commands, where appropriate. You may have no trouble understanding 250PRINTT5 or 300FORI5 = PTOM

but a reader will find 250 PRINT T5 and 300 FOR I5 = P TO M much easier to copy or edit. Many BASICs *do* insist on the spaces; the new IBM Personal Computer is one that does. Your computer may allow a larger program or run faster if you delete spaces and remarks, but you make the program much harder to transport when you delete them. It may be worth keeping two programs, a transportable copy and a condensed, quick-run copy.

3. Avoid unprintable characters. Where a few are necessary, find a way to make their presence visible. For instance, a disk read in Applesoft requires that you PRINT a CONTROL-D followed by a string. You can make this readable by

```
200 D$ = CHR$(4) :REM CONTROL-D
...
540 PRINT D$;"OPEN FILENAME" :REM DOS
    COMMAND STARTS CTRL-D
```

It is a good idea to indicate what other CHR$ characters are when they are created, too – for instance when CHR$ is used to put a quote mark into a string, or manipulate carriage returns or line feeds.

4. Identify specific features you depend on. This happens most often in connection with PRINT and INPUT statements. Most of us can guess what someone else's PRINT statements are supposed to do, but the INPUTs are another matter.

Some systems input a sentence like "TODAY IT RAINS" by INPUT A$ and the response ?TODAY IT RAINS; others by INPUT A$ and response ?"TODAY IT RAINS"; others by INPUT LINE A$ or by LINPUT A$ or even by INPUT (FIELD 40) A$. You can make this clear to the reader – so that he can try to do the appropriate thing on his system – by remarks, but clear user instructions within the program are probably even better. For example,

```
110 PRINT "TYPE IN A SENTENCE SURROUNDED
    BY QUOTE MARKS"
120 INPUT A$ :REM SAMPLE "HELLO, JOE,
    WADDAYA KNOW."
```

5. Make cues to the user extremely clear. Remember that you won't be around to show people how to use it; in fact, *no* expert on the program will be around. Give sample answers whenever possible, and protect against invalid answers.

```
130 PRINT"DO YOU WANT TO PLAY AGAIN (
    Y/N)";
140 INPUT A$
150 IF A$="N" THEN 9999
160 IF A$<>"Y" THEN 130
```

Note that invalid answers will cause the question to be asked again.

Next month, examples of portable program structure, input-output, and graphics programming.

Part II:

Writing Transportable BASIC

Edward Ordman

This concludes a two-part article on writing BASIC programs so that they are more easily read, revised, or translated to run on different computer brands. Though not everyone will agree with the goal (general-purpose BASIC), or the approach (structured programming), many of these suggestions are potentially useful to those programmers who later revise and improve their own programs. For contrast, see the views of some of the programmers quoted in "How The Pros Write Computer Games," elsewhere in this issue.

Structure

The major tool in making a program transportable is careful attention to program structure. This does not mean slavish adherence to "structured programming." It does mean using common sense and some of the important tools available to keep programs from becoming "spaghetti bowls" of GOTOs. This can include "structured programming" when applicable.

To consider a concrete example, suppose we have two branches in our code governed by a GOTO. A simple version might be:

```
500 IF X>2 THEN T=T+Y : C=C+2 ELSE T=T+Z :
    C=C+1
```

There is certainly no objection to writing this in one line if your BASIC allows it; the intent is clear. Remember that you should leave space for new lines, since someone may have to rewrite this as:

```
500 IF X>2 THEN 504
501 T = T+Y
502 C = C+2
503 GOTO 507
504 T = T+Z
505 C = C+1
507 REM ENDIF
```

Even this is still quite readable. It is clear where the IF starts and where its effect ends. A far worse example (but painfully common in beginners' programs) would have IF X>2 THEN 4000 and then down at line 4000 would have:

```
4000 T=T+Z: C=C+1 : GOTO 510
```

This is hard to read: how, when checking line 4000, can you know where it relates to the rest of the program? Reading lines 500-510, how can you understand the options of the other path?

My own practice, incidentally, is to avoid GOTOs over long distances, avoid upward GOTOs unless they are part of a fairly formal structure, and have GOTOs go to REM statements in a great many cases. Suppose, in the example above, line 510 was PRINT TAB(C);X;TAB(C+5); Y and some variation in the new machine meant that this had to be expanded to two lines to get the right spacing. A GOTO 510 in line 503 means that a line 509 cannot be introduced without other changes; the 507 REM means changes in the PRINT do not require changes in the IF.

A similar situation arises in programs where there is a large loop (PLAY AGAIN in a game) and some initialization before it. If you start

```
1 PRINT "WELCOME TO THE GAME"
2 T = 0
3 X = RND(1)
4 Y = 1
```

the person rewriting this may type 2 T=0: X=RND(1): Y=1, and be in big trouble when he discovers that at line 5560 you have GOTO 4. He will be in more trouble when he revises the program and needs to add another statement within the main loop, but before Y=1. Compare the program:

```
10 PRINT "WELCOME TO THE GAME"
20 T = 0:X = RND(1):REM INITIALIZE,0<
   X<1
30 REM  ENTER MAIN LOOP HERE
40 Y = 1 :REM  COUNT NUMBER OF ATTEMP
   TS
5560 GOTO 30 :REM REPEAT MAIN LOOP
```

In this version, the rewriter will not confuse line 20 and line 40; a line 35 can be added; and there is no confusion as to exactly where the GOTO is leading, even after several program revisions. In general, do not GOTO "the middle" of a line of reasoning without clearly labeling why and providing an easy way to make changes without extensive rewriting.

If you really want to avoid upward GOTOs in as many cases as possible (and it *does* make programs easier to read!), there are two alternative structures that are important: GOSUB ... RETURN and the DO ... WHILE. First, let us consider the DO...WHILE.

DO...WHILE can be regarded as an extension of FOR...NEXT. A typical form is:

```
1000 DO WHILE X>10
1010 PRINT X
1020 T = T+X
1030 X = X/2
1040 ENDWHILE
```

Suppose X is 50 when this is entered. Lines 1010-1030 will be done for X=50, for X=25, for X=12.5; then X will become 6.25, the test will fail, and the program will go on after line 1040. This is a remarkably useful *thinking* tool, even if your BASIC does not have these statements (many do not). But, for transportability, I would argue *against* using these statements even if you have them. There is, however, no reason at all not to *think* in terms of DO...WHILE and then to write an imitation of it:

```
1000 IF X<=10 THEN 1040 :REM DO WHILE
      X>10 TO LINE 1040
1010 PRINT X
1020 T = T+X
1030 X = X/2
1035 GOTO 1000
1040 REM  END WHILE
```

Again, this is easy to read, the upward GOTO is clearly explained, and the reader is in no doubt as to the scope of the loop and where you enter and leave it.

Subroutines Are Best

Subroutines – the facility provided by GOSUB and RETURN – are the single most important feature in providing transportability. There is a strong case to be made for dividing every program of more than a few dozen lines, and many shorter ones, into subroutines. Ideally, each subroutine should have a purpose that you can describe in one or two lines, and that explanation should be given in remarks at the head of the subroutine. The subroutine should *not* interact with the rest of the program except as provided in the leading remarks. An example:

```
6000 REM  SUBROUTINE TO CONVERT TO PO
     LAR COORDINATES
6001 REM  GIVEN X,Y COORDINATES.  RET
     URN R=RADIUS,T=ANGLE.
6002 REM  X,Y UNCHANGED.  RETURN T=0
     IF R=0.
6003 REM
6010 R=SQR(X*X + Y*Y)
6020 IF R = 0 THEN T = 0 : RETURN
6030 IF X<>0 THEN T = ATN(Y/X) : RETU
     RN : REM ARCTANGENT, RADIANS
6040 IF Y > 0 THEN T = 3.14159/2
6050 IF Y < 0 THEN T = -3.14159/2
6090 RETURN
```

It is entirely appropriate for subroutines to call other subroutines, or for a main program to consist primarily of subroutine calls, with all the real work done in the subroutines. But when this is done, it is even more important to make sure that the subroutines can be debugged separately – that they do not, for instance, change the variable used elsewhere, but not mentioned in the headnote.

Where you are using a feature that you know is particular to your computer – for instance, disk input/output – it is especially important to isolate it in a subroutine, and label it as machine-dependent. This means that it can be rewritten later with a minimum of change to the main program logic.

Make Input/Output General

It is very likely that anyone rewriting a program for another machine will have to revise input/output statements. This applies to PRINT and INPUT for keyboards, terminals, CRTs, and printers; to cassette and disk storage; to game controllers and joysticks; and to all other peripherals. Essentially the only "minimal" features that all machines have in common are INPUT X and PRINT X,Y,Z, and even these are not as standard as one might like. The usual solution is to stick to minimal formatting, if you consider transportability of prime importance; or to place fancy input/output in subroutines and indicate your intention clearly, if it is essential to the program. Here we can give only a quick guide to some of the tricks and pitfalls.

INPUT Some computers allow you to cue the user (prompt) as desired, e.g., INPUT "YOUR NEXT GUESS?";N while others do not. The others can fake it by PRINT "YOUR NEXT GUESS";:INPUT N getting the question mark on the same line as the printout. Many BASICs will not allow suppression of the question mark. Inputting string variables, particularly with embedded spaces or commas, also differs dramatically from system to system, as mentioned earlier. If your program depends heavily on a precise form of string input, place the input routine in a subroutine and explain the purpose carefully. For example:

```
2000 REM  STRING S$ WILL BE ALL CHARA
     CTERS TYPED (PRINTABLE OR NOT)
2001 REM  UNTIL ENTER IS HIT (EXCLUDI
     NG THE ENTER)
2010 S$ = ""
2020 K$ = INKEY$ :REM  GETS SINGLE KE
     Y FROM KEYBOARD
2030 IF K$ = "" THEN 2020
2040 IF ASC(K$) = 13 THEN 2090 :REM
     CARRIAGE RETURN, OR ENTER
2050 S$ = S$ + K$
2060 GOTO 2020
2090 RETURN
```

Of course, other machines may require substantial rewriting of this subroutine, if the special word INKEY$ is not available or works differently. In some microcomputers, the implementation may be as easy as INPUT LINE S$. Still, having this in a single subroutine, rather than scattered throughout the program, will simplify the job of rewriting for a new machine.

PRINT Some computers allow statements like PRINT "$"X"000", without commas or semicolons, and produce the output $4000 when X is 4. Others require PRINT "$";X;"000" and produce $ 4 000 or something similar. Usually, a clear indication (in a REMark) of what you want is far more helpful than an ingenious trick to achieve it on your machine. The exact meanings of comma and semicolon differ from one machine to another: it is universal that comma means "wide space; arrange in columns" and semicolon means "short space, or no space," but the details differ. In many configurations, TAB will not work properly (this is common when using a printer attached to an Apple, for instance).

If you must engage in any fancier spacing than use of commas and semicolons, explain yourself in REMarks and leave it to the reader to implement it on his machine. Many microcomputers do not have PRINT USING; if you use it, an example output line contained in a remark is very helpful. If you use a fancy PRINT statement repeatedly in your program, consider placing it to a subroutine where the reader will have to translate it only once.

CLEAR There are a number of special commands whose implementation differs from one computer to another. Some examples are Clear Screen, Go to top-of-page, and similar ones. (Varying print character width, for instance, is usually a function of the printer model, not of the BASIC.) If at all possible, place these functions on a line by themselves and remark clearly; it will then be easy for the reader to translate them, or delete them if inapplicable to the new system.

Joysticks These also differ dramatically from one system to another. Again, place them in a clearly labeled section of the program, preferably a subroutine, and label what they do. In particular, avoid repeating these statements numerous places within the program. Example:

```
1050 GOSUB 5000 : REM  READ PADDLES
2300 GOSUB 5000
5000 REM  READ PADDLES X,Y -- VALUES
     ARE 0 - 255, SCALE TO 0 - 100
5010 X = PDL(0)/2.55
5020 Y = PDL(1)/2.55
5030 RETURN
```

Clearly, someone whose paddle-reading commands are different, or give values in a different range, can easily rewrite this subroutine.

Tape/Disk While the particular statements involved in tape and disk input/output differ for almost every system, the general functions to be performed are almost identical. Typically, one must specify a file name and number by which it will be referred, and whether it will be for input (READ or INPUT), or output (WRITE or PRINT), or both. A typical statement is something like OPEN "DATAFILE" AS 1 FOR INPUT. If your BASIC allows omitting some of this, include it in a REMark. For example,

```
1050 PRINT D$; "OPEN INPUT";F$ :REM OPEN F$,
     SEQUENTIAL, INPUT ONLY
```

is acceptable if you only have one file open at a time; the reader can insert an AS #1 if the new system requires it. Once you have opened a file, you must read from it or write to it, typically by a statement such as READ #1, A,B,C or PRINT #3,A;C$;B;C$;X :REM C$=",".

Notice that if your system does not require a specific indication in the statement that it refers to a file, you should include one in a REMark. It is an excellent idea to write commas as field dividers to a file, even if your system will permit a space as a divider on input. Enough systems insist on the comma that it decreases portability to omit it. A statement such as

```
1060 REM A TYPICAL LINE OF FILE IS 4 , 5 ,
     DEBITS , 2.95 (CR)
```

will often make the program much clearer to the reader than it is from just the line

```
INPUT#3 P(K),Z(K),D$,A(I)
```

In the case of a direct access file, most systems also need to know the record length and record number for each read or write. If a direct access file is opened for updating, you should read a record before you write it. Finally, on any type of file, you should remember to close it explicitly (usually CLOSE #3 or some variant). Even if your BASIC does not insist on this, someone else's will; and it can be hard to figure out *when* to do the closing in a strange program.

A program using no files is more easily transportable than one using files; the fewer the files, the more transportable. (Avoid opening more files than needed at one time.) Sequential files are easier to move than direct access files; files read or written "all-at-once" are more transportable than ones that are read or written only intermittently. If at all possible, structure a program like this:

```
1000 GOSUB 7500 : REM READ WHOLE FILE
      INTO AN ARRAY
 .... :REM  MAIN PROGRAM ACTS ON THE
ARRAY
4000 GOSUB 7700 : REM WRITE WHOLE ARR
      AY BACK OUT TO FILE
4010 GOTO 9999
```

so that all file-handling is confined to specific subroutines and the files can be kept on a cassette tape even without fancy automatic stop-start features.

Graphics

If we view BASIC as something almost geological, something that has had layers added over time, graphics capabilities are the last layer, and the layer least solidified. Graphics differ more from machine to machine than any other feature. Fancy graphics tricks are the very hardest thing to transport from one system to another. Still, it is possible to do some graphics work and still limit the problems when moving them to another system.

Generally, it is easiest to transport a program that uses only "character" graphics. If we view the screen as consisting of a fixed number of rows and a fixed number of columns, then each position can be occupied by one letter or "character." If we confine ourselves to commonly available characters, our program *should* be capable of being rewritten for most systems. If it does not involve moving pictures, it should even be possible to run it on a printer-oriented system in many cases.

As you know, common systems *do* differ in screen size (in number of characters in a row or column). The first thing we must do is let the reader know what assumptions we have made:

```
50 M1=16:REM NUMBER OF LINES ON SCREEN
60 M2=40 :REM NUMBER OF CHARACTERS PER
   LINE
```

From this point on, we should place everything in terms of the numbers M1 and M2, *not* 16 and 40. Further, to position a given character C$ at coordinates X,Y (that is, X across and Y down: position X of row Y), we should set X, Y, and C$ and then call a subroutine. On the IBM Personal Computer, we print an "A" in the center of the screen by

```
100 X=INT(M2/2) :Y=INT(M1/2) :C$="A"
110 GOSUB 7000
 ....
7000 REM  SUBROUTINE TO WRITE C$ AT
     POSITION X OVER, Y DOWN ******
7010 LOCATE Y,X : PRINT C$;
7020 RETURN
```

Again, the user of any given computer can rewrite this subroutine as a whole far more easily than he can rewrite statements like LOCATE 12,40: PRINT "A"; which are scattered throughout the program.

Sometimes a screen is built up by "jumping around," rather than line-by-line. If you wish to get hard copy of such a screen, and lack a built-in operating system procedure to do so, you can have the subroutine just mentioned build an array by 7015 S(X,Y)=ASC(C$) (or 7015 S$(X,Y)=C$) and later print the entire array. This may be as easy as:

```
8000 REM  PRINT THE SCREEN STORED IN
     ARRAY S(M2,M1) *************
8010 FOR I = 1 TO M2
8020 FOR J = 1 TO M1
8030 PRINT CHR$(S(J,I)); :REM ; LEAVE
     S NO SPACE ON IBM PERSONAL COMP
8040 NEXT J
8050 PRINT :REM  GO TO NEXT LINE - DE
     LETE IF IT CAUSES DOUBLE SPACING
8060 NEXT I
8070 RETURN
```

Note that this program must contain a line such as

```
70 DIM S(80,24) :REM SAVE SCREEN. NOTE DIM
   S(M2,M1)
```

so that a person changing M1 and M2 will know how it changes the DIM statement.

A remarkable assortment of graphics effects may be achieved just by the skillful use of standard characters: minus signs or underscores for horizontal lines, ones or a special symbol for verticals, and so on. It is not hard to generate pictures by hand: hold a piece of window screen over a picture, judge the amount of darkness as best you can (most people can rate "darkest, dark, middle, light, clear") and use characters such as M I : . and space to represent them. Some scaling may be needed; in many systems the space allocated for a character is 1 2/3 to 2 times as tall as it is wide. Fill-in-the-blanks effects, on screen or paper, may be achieved by using minus signs as underscores:

SOCIAL SECURITY NUMBER ? ---–--–---

Turning now to "high-resolution" graphics, or other extended graphics features, we find that most of them still can be expressed in terms of X-Y coordinates and making a specific mark at specific coordinates, although the mark is now usually "on" or "off" or "COLOR 7" instead of a letter. The same principle as before applies: specify the maximum size involved; if at all possible give dimensions as fractions of M1 and M2 rather than absolute numbers; and keep the actual writing in as few subroutines as possible.

In general, have one subroutine that draws a point; another that draws a line by making repeated GOSUBs to the subroutine to mark points; and so on. Even if your computer has built-in line-drawing commands, place them in subroutines (instead of HLIN 20,50 TO 30,40 write X1=50: Y1=20: X2=40: Y2=30: GOSUB 2600 where 2600 has the line HLIN Y1,X1 TO Y2,X2), so that a person whose computer lacks them can try to write a reasonable imitation.

If you write carelessly, or depend too heavily on features of a particular machine, you can have a program that is very hard to translate to any other machine. If you want to be able to move your programs to a new, different machine, or have them run on a friend's machine or on a machine at school, you must plan ahead when you first write the program.

It takes relatively little extra effort to write a transportable program, and there are many fringe benefits. You yourself will find the program easier to test, debug, or reread a few months later. A little avoidance of particular machine "special features," a little use of good structuring practices, and some care to isolate likely-to-change features in labeled subroutines, can pay off in far easier maintenance and rewriting. And if it means that some published programs will run on a larger variety of machines than they used to, it will pay off for all of us. ©