A Tale of Four BASICs

which one for your 6800?

Rich Didday
1218 Broadway
Santa Cruz CA 95062

Rich has provided us with a rather gory look at what you'll have to go through if you're foolish enough (as he was) to try implementing foreign BASIC interpreters on your machine. But, the primary intent of his article is to provide a review and comparison of four popular BASICs for 6800-based machines . . . and he does an excellent job. I'd like very much to see other articles along this line comparing 8080-based BASICs, assemblers, text editors and others. — John.

I have four different versions of BASIC up and running on my 6800-based system, and I have the feeling that I might have a few more before I'm through. I'm not collecting BASIC interpreters as if they were old coins or something — it's just that I want to find a version that really suits me. What am I looking for? I want a version that: 1. will run any program written in ANSI Minimal BASIC (see Box 1); 2. is convenient to use; 3. is reasonably fast; 4. is reasonably inexpensive.

Besides that, I've just been curious about what sort of software products is becoming available.

I thought I'd share my experiences with you, so that if you're thinking of buying a version of BASIC, you might find some of your questions answered here — before you send in your hard-earned cash.

There's a wide range of differences among the four versions, and, depending on your priorities, almost any one might be best for you. I'll cover every aspect I can think of, from cost to speed to documentation to numerical accuracy to range of statements provided, etc.

The Four BASICs

Table 1 shows the four BASICs, with the latest prices I could find. I've listed them, and I'll discuss them, in historical order of their appearance on my system.

As you may have guessed by looking at the price, I didn't purchase Sphere's provisional nonextended BASIC — it came with my system. It was the first version I had up and running, the one I've used the least, and the main force behind my search for other versions.

Next I ordered a copy of Pittman Tiny BASIC; it showed up immediately (less than two weeks). Since it's available only on paper tape, I had to get my friend Nick

Version.	Manufacturer.	Cost for paper tape.	Cost for cassette (KC Std.).
Sphere provisional nonextended BASIC	Sphere Corp. 791 South 500 West Bountiful UT 84010	\$325[a]	\$300[a]
Pittman Tiny BASIC	Itty Bitty Computers P.O. Box 23189 San Jose CA 95153	\$5	
SWTPC 8K BASIC Version 1.0	Southwest Technical Products Corporation 219 W. Rhapsody San Antonio TX 78216	\$20	\$9.95
TSC Micro BASIC Plus Version 2.1	Technical Systems Consultants Box 2574 W. Lafayette IN 47906	\$6 + \$15.95[b]	\$6.95 + \$15.95[b]

[a]This is the price listed for "BASIC Version 1" in the June 1976 *Global News* (Sphere's newsletter). I believe that this is the "provisional nonextended BASIC" that comes with the Sphere 330.

[b]The documentation for TSC Micro BASIC Plus is priced and sold separately.

Table 1. The four BASICs.

BASIC is a higher-level, interactive computer language that was originally developed at Dartmouth College in the middle 1960s. The design goal was to create a reasonably powerful language that would be particularly easy for beginners to learn and use.

It is easy to learn, and it is also easy to implement. That is, compared to other higher-level languages, it is relatively simple to write a program that accepts statements in BASIC and carries them out. This program (which is a machine-language program tailored to the system it is to run on) is called an *interpreter.* An interpreter differs from a *compiler,* which is a program that accepts statements in some higher-level language and translates them into a machine-language program that can be run later to carry out the instructions of the original higher-level program. An interpreter, on the other hand, translates each statement of the higher-level program each time it comes to it, and carries it out immediately.

If you buy one of the four versions of BASIC discussed in this article, you get a copy of a machine-language program (i.e., the interpreter) plus documentation that describes what you got, how to tailor it to your system and how to use it. More abstractly, you also get the capability to enter, alter and run a broad variety of programs in some variant of the higher-level language BASIC.

There are, then, two different sorts of questions you need to answer in order to decide which version is best for you.

1. Questions about the interpreter, considered as a machine language program in and of itself. For example: Will the interpreter run properly on your system. Will it be hard to adapt to idiosyncrasies of your system (will it be hard to install)? How much memory space will it need? What's the cost?

2. Questions about the language the interpreter accepts. For example: What statements are allowed? What data types are allowed? How fast will a given program be carried out? How convenient is it to enter, alter and run programs using this interpreter? How useful are the error messages?

The answers to these questions are not all simple. Here's just one example. You might think that the more statement types a version of BASIC lets you use, the better; or that the more statements stolen from FORTRAN or ALGOL or whatever your *real* favorite language is, the better. In fact, every vendor sticks fancy little things in his version to try to attract and hold you. But what if one of your goals is to be able to trade programs with other people? What if you don't want to do much programming yourself, but want to buy programs to run on your system? Then anything but the fairly well-agreed-on, "standard" statements in your BASIC will be useless.

This factor itself would be easier to judge if there were an agreed-on standard version. The American National Standards Institute (ANSI) has been working on a set of standards for what they call "Minimal BASIC" for some time now. (A draft of the proposed American National Standard Programming Language Minimal BASIC, report BSR X3.60, X3J2, January 1976, is available from: CBEMA/Secretary X3, 1828 L St. NW, Washington DC 20036.) Their plan is to define a minimal subset, and then define a series of "enhancements" covering such things as string manipulation, matrix operations, file manipulation, etc. When they finally agree on the standards, we'll be in better shape — you'll be able to tell exactly what's included in a version if it says "ANSI Minimal BASIC with string enhancement." Right now, you're on your own! Good luck.

Box 1.

to make a copy on cassette tape. I got it flying with relatively little effort, and have used it to write a fairly lengthy game program. It's a carefully done piece of software, and I'd use it more if I didn't care about being able to write programs that conform to the proposed ANSI standards.

The third version was Southwest Tech's 8K BASIC. I knew when I got it that it was made specifically for the SWTPC 6800 machine, that it didn't come with a source version of the interpreter, and that it would certainly be a lot of work (or maybe even impossible) to convert it to my system; but I felt like a challenge, or else I was desperate (I don't remember which). As it turns out, it's the version that I use most.

The fourth is TSC's 6800 Micro BASIC Plus, which, as the name implies, lies between Tiny BASIC and a full version like SWTPC's 8K in size and capabilities. It arrived instantaneously (less than one week).

Documentation

There's quite a range here, all the way from virtually nothing up to a 68-page booklet describing the product inside and out (see Table 2).

The documentation provided with Tiny BASIC, SWTPC 8K, and Micro BASIC Plus is generally very good. Each gives thorough descriptions of each statement type. They're all good about giving little examples of the use of each construct, they all list error message meanings, and they all give lists of the memory addresses of important stored variables (start of user program, entry points to the interpreter, end of memory pointer, etc.).

I've programmed in BASIC for quite a while, so it's hard for me to be certain, but I *think* someone who had never programmed before might be able to pick up enough from the Tiny BASIC documentation to be able to write programs. The others assume that you know something about programming, and I guess Sphere assumes you're clairvoyant.

Range of Statements Provided

Table 3 shows the statements that are provided by each of the four versions. You can probably figure out what most of them do. If not, see Table 1 in Stephen Pereira's "Now It's Imsai BASIC!" *Kilobaud* No. 5, May, 1977.

Though all four BASICs have quite a few statement types in common, there are some slight differences. All provide a limited text-editing capability, which lets you enter (numbered) lines, list

Version	Number of pages of documentation	Contents
Sphere provisional nonextended BASIC	3	A list of legal keywords and operators plus two pages telling how to load the tape.
Pittman Tiny BASIC	26	Clear descriptions of statement types and meanings, three pages of sample programs, instructions for installation on seven different systems.
SWTPC 8K BASIC Version 1.0	26	Clear descriptions with small examples of all statements.
TSC 6800 Micro BASIC Plus, Version 2.1	68	Clear descriptions of all statement types with examples, lots of useful little side notes, plus complete listing of assembly language source text (well commented).

Table 2. Documentation.

them to see what you've got, and wipe out or alter existing lines. Since many, if not most, home users will be employing TVs or other video displays, it's important to be able to list only a specified number of lines. The typical form for doing this is: LIST 30,120 — which lists all lines from number 30 up to and including number 120. All except Sphere allow this form. As nearly as I can tell, there is no way in Sphere's version to have a look at the first part of a long program. Forms like LIST 30 are allowed, but list all statements from line 30 through the end of the program. Even though Sphere does everything slowly, it doesn't list slowly enough for you to read the program on the fly.

Every version provides a way to wipe out the program you've been working on and start a new one. In Sphere and TSC, you type SCRATCH. In Tiny BASIC, you type CLEAR; and in SWTPC 8K you type NEW.

Two versions have a command that returns control to the host computer's monitor. PATCH does this in SWTPC 8K; MONITOR does it in Micro BASIC Plus. Tiny BASIC doesn't have a separate command for it, but it's not hard to put together two calls to the USR function (which carries out machine language routines for you) to accomplish it (and Pittman's documentation makes it perfectly clear).

In Sphere, the only way to get back to the monitor (or even to just interrupt an executing program) is to (shudder) reset the machine. On my system, that means you're flirting with disaster, since hitting the reset switch stops the dynamic memory refresh. After a while, you get pretty good at getting your finger off the reset button quickly, but still . . . .

Both TSC and Pittman tell you carefully and completely what to do if you don't like their choices of breakpoint, back space and prompt characters. Specifically, they tell you where those characters are stored, so you can insert your own choices if you wish. SWTPC doesn't give you that information, and you're stuck with what they give you. Sphere, believe it or not, doesn't even have back-space capability. If you make a mistake in typing, there's no way to go back; you just have to hit return and wait for the thing to give you an error message!

SWTPC 8K comes closest to being able to run any program that conforms to the (soon to be adopted) ANSI standards for Minimal BASIC. SWTPC's documentation points this out and notes one disagreement with the standards — all arrays start at location 1 in SWTPC 8K. (In the proposed standards, it is the programmer's option whether arrays start at 0 or 1.) I haven't made a detailed check of SWTPC 8K against the standards, but since it is claimed that the array starting point is the only conflict with the standards, I'll mention a few other conflicts I chanced across.

The first *might* simply be a bug I introduced when I installed the interpreter on my system. At any rate, according to the proposed standards,

```
20 PRINT "LINE 1",
30 PRINT "AGAIN"
40 END
```

Sphere provisional nonextended BASIC

Commands	Statements	Functions	
RUN*	DATA	ABS	
LIST*	DIM	ATAN	
SCRATCH*	DEF	COS	
	END	EXP	
	FOR NEXT STEP	INT	
	GOSUB	LOG	
	GOTO	RND	
	IF THEN	SGN	
	INPUT	SIN	
	LET	SQR	
	MAT	TAN	
	PRINT		
	READ	CON	
	REM	IDN	
	RESTORE	INV	matrix operations
	RETURN	TRN	
	STOP	ZER	

TSC Micro BASIC Plus

Commands	Statements	Functions
RUN*	DATA	ABS
LIST*	DIM	RND
SCRATCH*	END	SGN
MONITOR*	EXTERNAL*	SPC
	FOR NEXT STEP	TAB
	GOSUB*	
	GOTO*	
	IF THEN	
	INPUT	
	LET*	
	ON GOSUB	
	ON GOTO	
	PRINT*	
	READ	
	REM	
	RESTORE*	
	RETURN	

SWTPC 8K BASIC

Commands	Statements	Functions	
RUN*	DATA	ABS	
LIST*	DEF	COS	
NEW*	DIM	EXP	
PATCH*	END	INT	
SAVE*	FOR NEXT STEP	LOG	
LOAD*	GOSUB*	PEEK	
APPEND*	GOTO*	POS	
LINE*	IF THEN*	RND	
DIGIT*	INPUT	SGN	
	LET*	SIN	
	ON GOSUB*	SQR	
	ON GOTO*	TAB	
	POKE*	TAN	
	PORT*	USER	
	READ*		
	REM	ASC	
	RESTORE*	CHR$	
	RETURN	LEFT$	
	STOP*	LEN	string operations
		MID$	
		RIGHT$	
		STR$	
		VAL	

Pittman Tiny BASIC

Commands	Statements	Functions
RUN*	END*	RND
LIST*	GOSUB*	USR
CLEAR*	GOTO*	
	IF THEN*	
	INPUT*	
	LET*	
	PRINT*	
	REM*	
	RETURN*	

Table 3. The commands, statement types and functions provided by the four BASICs. Those which can be used in the direct mode, i.e., without being part of a program, are marked by an asterisk.

```
1000 PRINT "I ";
1010 FOR S=1 TO -1
1020   PRINT "DON'T ";
1030 NEXT S
1040 PRINT "CONFORM TO THE ANSI STANDARDS"
1050 PRINT "CONCERNING FOR LOOPS."
1060 END
```

Example 1.

```
I DON'T CONFORM TO THE ANSI STANDARDS
CONCERNING FOR LOOPS.
```

Example 2.

should yield a single output line when run, like this:

```
LINE 1              AGAIN
```

but (on my system at least) the program prints on two lines:

```
LINE 1
AGAIN
```

Another conflict with the standards concerns FOR loops. Running the program in Example 1 in SWTPC 8K produces the output in Example 2.

If the standard had been followed, the "DON'T" would not have been printed. This is because the initial value of S is already greater than the upper limit (-1).

Finally, the proposed standards decree that the random number generator function (RND) not take an argument, and that a command RANDOMIZE, which initializes RND, be included. In SWTPC 8K, RND must be given an argument, as in

```
LET X=RND(0.0)
```

and RANDOMIZE is not implemented.

Overall, though, SWTPC 8K is the only one of the four that comes close to the proposed standards.

Sphere comes next closest to being able to run any legal ANSI Minimal BASIC program; neither Tiny BASIC nor Micro BASIC Plus makes any real pretense of trying to conform to the proposed standards. Their design strategy calls for providing as much power for the user as possible within a severely limited memory space, and that leads both of them away from standard forms of BASIC. If your system has enough memory to run something like SWTPC 8K, you can decide how important it is for you to be able to run standard BASIC progams. If you have only 4 or 5K of memory, having a full Minimal BASIC is a frill you can't afford.

Sphere's version does have some matrix operations (see Table 3). For example, these statements

```
10 DIM A(20,20)
20 MAT A= IDN
```

store the identity matrix (1s on the diagonal, 0s everywhere else) in the array A. Interestingly enough, these statements

```
10 DIM A(20,20), B(20,20)
20 MAT A= INV(B)
```

do the same thing, no matter what B is. The documentation says that INV stands for inverse, but it has the same effect as IDN. Oh, well.

SWTPC 8K includes a number of string operations, and allows arrays of strings. I personally find these features extremely useful. SWTPC also incorporates the commands SAVE, LOAD, APPEND, which (respectively) store an active program on tape, load a program from tape and append more lines to an existing program in the machine. None of the versions includes an explicit way to store and retrieve data on tape — SWTPC makes it reasonably easy, though.

Memory Requirements

There's a big variety in the memory requirements of the four BASICs. Obviously, versions (like SWTPC) that include many different legal commands are going to require more memory than those with a more limited repertoire (like Tiny BASIC). If your main constraint is limited RAM, Table 4 may be enough for you to choose which of the four is right for you.

In terms of memory (as well as other requirements), Sphere is in a category all its own. Apparently, it's actually

Version.	Approx. memory space required for interpreter alone.	Suggested minimum memory.	Memory required for medium-sized programs.
Sphere provisional nonextended BASIC.	15.2K	20K	20K
Pittman Tiny BASIC.	2.5K	3K	4K
SWTPC 8K BASIC Version 1.0.	7.1K	8K	10K
TSC 6800 Micro BASIC Plus, Version 2.1.	3.3K	4K	5K

Table 4. Memory requirements of the four BASICs.

Version.	Internal representation.	Number of digits displayed.	Decimal Digits of accuracy.	Largest number.	Smallest number greater than 0.
Sphere	floating point, binary, 4 bytes per value	6	4 or less	5.793E76	4.0E-78[1]
Pittman	fixed point, binary, 2 bytes per value	5	4+	+32767	1
SWTPC	floating point, BCD, 6 bytes per value	9	9[2] or 6[2]	9.99999999E+99	1.E-99
TSC	fixed point, BCD, 3 bytes per value	5	5	+99999	1

[1] Sphere BASIC has a number system all its own — each different scheme I hit on to determine the smallest number greater than zero produces a different result, some with exponents of -78, others with exponents of -77.

[2] Normal arithmetic (+, -, *, /) yields results accurate to 9 decimal digits. The built-in functions (like SIN, COS, EXP, etc.) give results accurate to 6 decimal digits.

Table 5. Number representations in the four BASICs.

a batch (as opposed to interactive) version lifted (with a few alterations made and bugs added) from another machine. Instead of translating that version into 6800 machine language, Sphere wrote an emulator for the original machine. So, when you run Sphere's version, you're actually simulating running BASIC on another machine. While this *is* a clever way to bring up a version of BASIC on the 6800 rather rapidly, it requires so much memory, runs so slowly (as we'll soon see) and is so inconvenient to use that it is acceptable only as a stopgap measure. Since other versions are available, it seems that Sphere BASIC has little to recommend it.

Arithmetic

As you can see in Table 5, two of the versions store numbers in binary coded decimal (BCD), two use the more normal binary representation. The two larger versions offer floating-point numbers (numbers with exponents and fractional parts); the two smaller ones don't.

Both SWTPC and Sphere provide a range of arithmetic functions (SIN = trigonometric sine, COS = cosine, LOG = natural logarithm, SQR = square root, etc.). The SWTPC arithmetic functions are accurate to six decimal digits (operations like adding, subtracting, multiplying and dividing are accurate to nine decimal digits). SWTPC arithmetic functions are inordinately slow (more on this later). Most of Sphere's arithmetic operations and functions are accurate to four decimal digits (although they are displayed to six digits), but some of them are unbelievably inaccurate. For example, Sphere's LOG function has no digits of accuracy for arguments around 1.0 (see benchmark program functions in Table 6). And (see if you believe this one), 1/(-1) evaluates to .25!

All except Micro BASIC Plus implement the normal operator precedence rules. In TSC's version, arithmetic operations are performed left to right, unless parentheses are used to force another order. Thus, PRINT 1 + 2*3 prints the value 9 in TSC BASIC. It prints 7 (as you'd expect) on the other three versions.

Speed

There are quite a few factors that affect how fast a given program will run on different interpreters. For instance, you'd expect that, in general, BASICs that implement floating-point arithmetic with a large number of significant digits would run slower than versions restricting themselves to small integers. On the other hand, you'd expect that versions made to fit into a tiny amount of memory would be a bit slower than those that could afford to do more elaborate processing on the program before it's run. You'd expect BASICs that provide a large number of different statements to run a little slower, because it probably will take longer to decipher any given statement. And, of course, you'd expect a version that was run by simulating another machine would run many times slower than one written specifically for the 6800.

I tested the relative speeds by running nine benchmark programs. The first seven are those used by Tom Rugg and Phil Feldman in their recent article (see "BASIC Timing Comparisons," *Kilobaud* No. 6, June 1977). Table 6 shows the results of all nine benchmark programs.

Since my system runs with a slow clock, I've normalized all the resulting times. If your 6800-based system runs at full-rated clock speed (1 MHz), you should observe the times shown in Table 6.

There aren't really any big surprises in the times taken for the first seven benchmark programs. For each of them, Sphere is more than ten times slower than the slowest of the other three. Micro BASIC Plus seems to be a little faster than you might expect. One interesting point is that the standard BASIC assignment statement using the keyword LET, as in 40 LET A=0, runs faster than the nonstandard ones used by Rugg and Feldman, 40 A=0, at least in Tiny BASIC. None of the other versions would accept the nonstandard assignment statement. TSC's documentation says that the nonstandard form is OK, but on my particular system, using it yields obscure error messages.

Since Rugg and Feldman didn't run any benchmarks that tested numeric functions or string manipulations, I made up two additional benchmark programs. They're shown as Program 1 and Program 2.

The function benchmark (Program 1) measures the maximum absolute error encountered in the operations it goes through. Since

Time (in seconds) to run benchmark programs.[a]

	1	2	3	4	5	6	7	functions	strings
Sphere	166[b]	493	1325[c]	1207½[c]	1290[c]	4250[c]	6003[d]	8059[c]	- -
Pittman	- -[e]	41[f]/37[g]	61	62	83	284[h]	- -	- -	- -
SWTPC	15[b]	25	96	105	109	173½	204	6631[c]	47
TSC	9[b]	19½	48	51	61	109	223½	- -	- -

[a]The first seven benchmark programs are those used by Tom Rugg and Phil Feldman in their article in *Kilobaud* No. 6, pages 66-70. Times shown are normalized to show the time required if the 6800 was run at its rated clock speed (1 MHz). To get the actual times observed on my system, multiply by 1.45 (e.g., TSC 6800 Micro BASIC Plus actually took 9x1.45 = 13.5 seconds to run benchmark program 1 on my system). Times are believed to be accurate within ½ second.

[b]This version of BASIC will not accept the nonstandard assignment statement (e.g., 400 K=0) used by Rugg and Feldman; so LET was inserted as necessary (as in 400 K=0).

[c]Program was actually run for 100 interations; elapsed time was multiplied by 10.

[d]Program was actually run for 10 iterations; elapsed time was multiplied by 100.

[e]Pittman Tiny BASIC does not implement FOR-NEXT loops.

[f]Using nonstandard assignment statements (i.e., without LET).

[g]Using standard assignment statements (i.e., with "LET").

[h]Pittman Tiny BASIC has no arrays, so the DIM statement in Rugg and Feldman's benchmark program number 7 was replaced by a REM statement. Also, the FOR-NEXT loop in their program was replaced by the equivalent counting loop.

Table 6. Speed.

```
  10 REM :FUNCTION SPEED AND
  20 REM :ACCURACY BENCHMARK
  30 PRINT "START"
  40 LET T=0
  50 LET E=0
  60 FOR I=1 TO 1000
  70  LET S=EXP(LOG(I))
  80  IF ABS(S-I)/I<=T THEN 100
  90    LET T=ABS(S-I)/I
 100  LET R=1/I
 110  LET F=SQR(SIN(R) Λ 2 + COS(R) Λ 2)
 120  IF ABS(F - 1.0)<=E THEN 140
 130    LET E=ABS(F - 1.0)
 140  NEXT I
 150 PRINT "DONE"
 160 PRINT "LARGEST ERROR IN EXP, LOG=";T
 170 PRINT "LARGEST ERROR IN SIN, COS, SQR=";E
 180 END
```

Program 1.

```
  10 REM :STRING MANIPULATION BENCHMARK
  20 PRINT "START"
  30 LET A$="0123456789"
  40 LET B$="ABCDEF"
  50 FOR I=1 TO 1000
  60  LET C$=A$ + B$
  70  LET C$=LEFT$(C$,1)
  80 NEXT I
  90 PRINT "DONE"
 100 END
```

Program 2.

EXP(LOG(I))=I and SQR(SIN^2(X) + COS^2(X))=1, this is an easy test to make. It's also a fairly severe test, since you might expect that even if, for example, the LOG function is accurate to six decimal digits and EXP is accurate to six, that EXP(LOG( )) might be accurate only to five places or so.

SWTPC comes out well in the function accuracy test, T=1.0E-06 and E=3.0E-07, which means you can really trust SWTPC's arithmetic functions to six decimal digits. The corresponding error measurements for Sphere are T=.418655 and E=1.6307E-4. That means you can trust Sphere's EXP and LOG to no decimal digits, and you can trust SQR, SIN and COS to three decimal digits. This may be a bit overstated — the real problem is that Sphere's LOG function is worthless for arguments around 1.0. Outside that range, it seems to be accurate to three or four decimal digits.

In terms of speed on the function test, there's a bit of a surprise. Here, SWTPC is in the same ball park as Sphere! For the first seven benchmarks, SWTPC is from 10 to 30 times faster than Sphere. All of a sudden, in the function benchmark, it's less than twice as fast. This puzzled me, so I started playing around.

At first I hypothesized that SWTPC was just using particularly bad algorithms to compute the functions. This began to look likely when I discovered that I could write a program *in BASIC* that could compute the SIN function to nine-digit accuracy almost as fast as the built-in SIN could compute it to six places! (Mine takes 75 seconds to compute the sine of the 100 angles from -49 to +49 radians, while the built-in SIN function takes 72 seconds to do the same.)

Convinced that I was on the right track, I coded my routine in machine language (but using calls to SWTPC's add, multiply and divide routines), expecting to speed things up by a factor of 20 or 30. But my machine-language sine routine runs just 3.5 times faster than SWTPC's. This suggests that most of the time in computing the arithmetic functions is spent in the add, multiply and divide routines, with the overhead of moving the arguments around, looping and testing approaching insignificance. I haven't successfully isolated the problem — every test I've been able to think of has been inconclusive. I *am* suspicious of the use of the BCD representation, however. To get nine decimal digits of accuracy, SWTPC allocates five bytes for the fractional part of each number. Using normal two's complement binary representation, you need just four bytes (2^{31}=2 147 483 648). Not only does using BCD take up more space, but the arithmetic operations are harder to perform.

Each of SWTPC's arithmetic routines is surprisingly long — even the one for negating a value (which is next to trivial using two's complement). Their documentation claims that using BCD gives greater accuracy. Nonsense. TSC (which also uses BCD representation) at least says something true — it's easier to convert decimal numbers into BCD than into two's complement binary. But, so what? BCD makes sense for supermarket cash registers, where the memory required to store the conversion program exceeds the amount of memory needed to store the inefficient BCD numbers, but not in a general-purpose situation.

Ease of Installation

All four BASICs are written for the 6800 microprocessor. Of course that doesn't mean that they'll work properly on just any old 6800-based system. Some are specifically fitted to specific systems (Sphere's is intended to work on the Sphere 330 or 340 and no other machines; SWTPC's is intended to run on the SWTPC 6800 computer and no other; Micro-BASIC Plus is intended to run on a number of different systems; Tiny BASIC is designed to be usable on a very broad range of 6800-based systems). Given your specific system, you face a variety of potential trouble spots in trying to install a given BASIC. To name a few: Do you have RAM at the locations required by the interpreter? Are your I/O interfaces of the type and at the addresses assumed by the person who wrote the interpreter? Does your output device accept and properly interpret the control characters these versions tend to tack onto output? Etc., etc.

The conflict that caused me the most trouble is a battle for the use of the bottom page of RAM — on my system the ROM monitor uses part of the bottom page for temporary storage. With the 6800, unlike with the 8080, there's a legitimate reason for the hardware manufacturer to preempt part of the bottom page of memory — instructions that refer to the bottom page take up one less byte than those that directly access higher memory locations. If a manufacturer provides a large program in ROM, he'll save space, hence money, using the bottom page for storage. Of course, the software producer has to make sure his software package takes less space than the competition's; so, he'll also want to use the bottom page. Everything's fine except for the end user: you and me.

Table 7 shows about how long it took me to get the four versions working on my system. Of course, the numbers for Sphere and SWTPC can change radically, depending on which machine you have. If I hadn't had a Sphere system, the time required to get Sphere provisional non-extended BASIC working would have been substantial.

Note! Please do not write me asking for a copy of my kludged-up version of SWTPC

8K BASIC. You don't want it. It's not the right solution to the problem. Maybe if enough people with 6800-based non-SWTPC machines wrote to SWTPC, they would do something about it.

Pittman does the best job of explaining how to install his version. He even provides hints and sample programs for the necessary I/O drivers on seven different systems.

TSC says in their advertisements that their version requires continuous memory space from location 0 up through the end of everything (i.e., at the end of your BASIC program is space for data values; your program is stored after the interpreter itself). Since that condition isn't satisfied on my system (due to the conflict with locations used by my monitor), maybe I should have just given up there. But, instead, since a well-commented assembly-language source listing of the interpreter is included in TSC's documentation, I was able to go through and shift references to memory locations my monitor uses. That took about four hours.

Then I spent about two hours writing the I/O drivers, and I was ready to watch it fly. I typed in PRINT "HELLO" and was rewarded with HELLO. So far, so good. Then I typed PRINT 2 and got 2. Feeling confident, I typed PRINT +2 and was presented with a situation in which the processor ran wild, wiping out part of the interpreter, winding up in an infinite loop, endlessly cycling through part of high memory space. (Let me skip the gory details of how I was able to detect all that.) I spent some four or five hours tracking down where the problem was occurring (this phase would have taken much longer without the source listing of the interpreter). Finally, two things clicked into place, and I realized what was going on.

First, the problem didn't occur in one specific spot, but randomly in one of three consecutive instructions. Second, the carnage started at a consistent location, which happens to be the place my system jumps to respond to a hardware interrupt. Finally, I looked over the assembly listing and discovered what I might equally well have been able to see in a few minutes if I had just thought of the possibility earlier — certain tests in the TSC arithmetic routines have the side effect of turning off the 6800 interrupt mask! Milliseconds after that, my real-time clock (or *any* of my I/O interfaces) calls for an interrupt, causing the processor to branch to where the interrupt-handling routine should have been — and Kapow! OK . . . so out with the manuals to figure out how to tell all the PIAs and ACIAs not to request interrupts, grumble, grumble, and finally, after about 12 hours, Micro BASIC Plus was up and flying.

I've saved SWTPC 8K for last in this discussion of "ease of installation" because it's a special case. SWTPC wants you to buy an SWTPC 6800 computer to run their BASIC on — they make no claim that it'll work on any other system, and they supply no instructions for installing it on anything but an SWTPC 6800. That means you'd have to be a fool, crazy, or both to try to bring it up on any other system. I plead guilty, and promise I'll never do it again. But since I did it this once, I thought it might be interesting to run through some of the trouble spots.

Actually, it was a team project. My friend Nick did a couple of days' work to get things started. He deciphered the SWTPC cassette-tape format and figured out where to insert calls to I/O routines, then turned it over to me. At this point, we had a BASIC that gave us an error message for every line or command we typed in. It also put out a tremendous number (for my system) of extraneous control characters, so that the ready prompt looked something like:

```
##
####READY
```

But that was a secondary problem.

Armed only with a few old SWTPC newsletters and a lot of coffee, I plunged in. For a few days, I alternated between trying to track down the location where everything went haywire and just browsing through the 7K+ of raw machine code, trying to get some vague feeling for what was what. Let me recount one problem in some detail.

Arithmetic expressions weren't being evaluated properly, so I started tracking down what the interpreter did to evaluate them. I established that a particular return address was getting clobbered in the middle of processing the expression 1 + 2*3. Before a particular subroutine jump, the return addresses on the stack were OK. After it, one of them had been changed, changed to a value that was obviously wrong. So, I started looking at the subroutine to see if I could figure out (vaguely) what it did. I couldn't really tell, though, since it consisted of a bunch of subroutine calls. I jumped into the middle, set a break-

Version	Time to install	Comments
Sphere provisional nonextended BASIC	30 minutes	No problems — made to run on my specific system
Pittman Tiny BASIC	3 hours	No particular problem — just had to write I/O routines and test for break character routine.
SWTPC 8K BASIC	75 hours	SWTPC 8K BASIC is made to run on the SWTPC 6800 only. Since I had neither an SWTPC 6800 nor a source listing of the BASIC interpreter, I had to slog through raw machine code trying to find out why it kept blowing up, to find where the I/O drivers were called, locate references to memory locations that my monitor thinks are its alone, change control characters, alter stack pointer initializations, etc.
TSC 6800 Micro BASIC Plus	12 hours	As they say in their catalog, ". . . TSC 6800 programs require RAM starting at memory location 00 and continuing uninterrupted through the amount required by the program . . ." and, to repeat, there are some low memory locations that my monitor thinks it owns. It took me about six hours to go through the source listing to find all conflicts, decide where to move the offending memory assignments, make the changes, write I/O routines. Then it took another four hours or so to discover that TSC turns off the interrupt mask during some arithmetic operations. That's rude of them.

Table 7. Ease of installation.

SWTPC 8K BASIC uses a simulated stack (i.e., a stack that uses neither the 6800 stack pointer register nor the 6800 stack operation instructions) to save the contents of the index register from time to time. Naturally, there is a Push routine and a Pop routine, and a pair of memory locations to store the address of the current stack top. And, of course there's code that initializes the address at the beginning of everything. When installed on the SWTPC 6800 computer (or on any system using the MIKBUG monitor), this address is initialized to $A07F_{16}$. Since I have no memory at that location, I picked an address essentially at random in a region where I *do* have RAM. I picked 4800_{16}. That turned out to be a disastrous choice. Here's why.

The Push routine works like this:

```
    get current stack pointer
    decrement it twice (using the DECX instruction, which
    performs a full 16-bit decrement)
    store the contents of X in the place pointed to by the
    stack pointer
    return.
```

So, after carrying out the Push routine once, the value of the stack pointer will be $4800-2 = 47FE_{16}$. So far so good.

The Pop routine works like this:

```
    get current stack pointer
    copy the contents of the (two) bytes indicated by the
    stack pointer into the X register
    increment the second byte of the stack pointer twice
    (using the INC instruction, which performs an 8-bit
    increment)
    return.
```

Now. Suppose we've just returned from whatever dirty work we were up to and want to restore the index register to its previous value. We call the Pop routine; it restores the X register and *then leaves the stack pointer with the value 4700!* (Not 4800, as it should be.)

After the next Push/Pop sequence, the stack pointer will be at 4600, and soon it'll be down among our BASIC program, wreaking havoc!

Note that this is *not* a bug in SWTPC 8K BASIC. The author of the interpreter knew what he was doing, had determined that no combination of circumstances could require the use of more than $7E_{16}$ bytes in this particular stack, and so initialized the stack pointer to A07F. The cure for my case was simple. All I needed to do was initialize the stack pointer to 477F instead of 4800. If only I had had some way of knowing that beforehand!

This particular problem is just one of many. It's of no great interest in and of itself — it does show the sort of pitfalls awaiting you when you try to do things the wrong way. No person in his or her right mind would attempt to install a 7000-byte machine-language program on an alien system with no relevant documentation.

Box 2.

point, got back into BASIC, typed in my sample statement (PRINT 1 + 2*3), hit return, and then when the interpreter hit the breakpoint, I looked at the return address that was causing trouble. It was still OK. So, I moved the breakpoint later in the subroutine and repeated until the offending call revealed itself.

Then I looked at *that* subroutine. It too consisted of a bunch of subroutine calls, so I repeated the process. After several hours, I finally found the place where the return address was being clobbered. Box 2 describes what was causing the problem. Once I figured out what was wrong, the solution took about 30 seconds! Finally, I could evaluate arithmetic expressions. After just two days' hard work.

If I had it to do over again, I wouldn't. What I have now is a kludged-together, slightly unstable mess, which usually runs properly. If and when SWTPC issues a new, improved version of their 8K, I'll have to go through hours of work to bring it up on my systems — most of the time I've put in on this version will be wasted. Making absolute machine-language patches in a large program that is virtually undocumented is just the wrong way to go. Period.

Bugs — Theirs and Yours

Let's face it. Sphere provisional nonextended BASIC (the only version Sphere has ever made available, to my knowledge) is a disaster. The others have few blatant bugs, but there are some rough edges here and there. In Micro BASIC Plus, for example, if an array A has been dimensioned to be of size 25 (say), your program can refer to and store into all locations from A(-25) to A(25). Apparently just the absolute value of the subscript is checked. This can lead to some hard-to-discover bugs in programs that involve complex subscript expressions. In addition, if by chance you give an array a dimension of 99 (which the manual says is not legal, but the interpreter doesn't check for), no subscript checking seems to be done at all, thus giving you a method to wipe out other data, your program, the interpreter. . . . Another rough edge in TSC's BASIC is that leading zeros aren't suppressed on printed negative values.

Here's a rough edge in SWTPC's 8K: The test for string inequality is a little wacky if the string values being compared are of different lengths. This makes it a little awkward to get "SMITH" to come before "SMITHY" when you're putting a list of names in alphabetical order. (See Table 8 for a fix you can use that isn't too terribly slow.)

I've used Tiny BASIC frequently and haven't come across anything I'd be willing to call a bug *or* a rough edge.

Of the four, Sphere seems to do the best job of error checking the program as you enter it. Of course, there's no way to know what the error numbers mean since that wasn't included in the documentation. At any rate, Sphere is the only version that checks each newly entered line for syntax errors as it's entered. Micro BASIC Plus will complain if it can't identify the keyword of a newly entered line (but not about any deeper errors). SWTPC doesn't complain about illegal keywords on entry, but does print a question mark in front of them when you list your program. Tiny BASIC doesn't check the newly entered line at all, it just stores it. That's no help in writing programs, but it does mean that you can use the Tiny BASIC system as a sort of text editor — you could enter in some text, list it, correct it, make a hard copy listing, and then cut off the line numbers, I suppose.

All four versions give error diagnostics if an illegal statement is encountered during execution of your program. I ran some tests to judge how appropriate the error messages seemed to be — Tiny BASIC won most of my tests, although I suspect that Sphere would win if I knew what its error numbers meant.

Overall Conclusions

Sphere BASIC has the most bugs in it, and is the hardest to use. Pittman Tiny BASIC is the easiest to install on the widest range of systems (assuming you have some way to read paper tape). SWTPC 8K BASIC has the most features and comes closest to the proposed ANSI standards. TSC's Micro BASIC Plus runs the first five benchmark programs faster than any of the other three.

```
 10 PRINT "NAME 1=";
 20 INPUT S$
 30 PRINT "NAME 2=";
 40 INPUT R$
100 REM :COMPARE THE STRINGS
120 IF S$ <=R$ THEN 200
130 PRINT R$;" COMES BEFORE ";S$
140 PRINT
150 GOTO 10
200 PRINT S$;" COMES BEFORE ";R$
210 PRINT
220 GOTO 10
230 END

RUN
NAME 1=? ADAMS
NAME 2=? BRONSON
ADAMS COMES BEFORE BRONSON

NAME 1=? SMITHY
NAME 2=? SMITH
SMITHY COMES BEFORE SMITH

 10 PRINT "NAME 1=";
 20 INPUT S$
 30 PRINT "NAME 2=";
 40 INPUT R$
 50 IF LEN(S$) <=LEN(R$) THEN 120
 60 REM :R$ IS LONGER, SWAP 'EM
 70 LET T$=R$
 80 LET R$=S$
 90 LET S$=T$
100 REM :COMPARE EQUAL SIZED
110 REM :PARTS OF THE STRINGS
120 IF S$ <=LEFT$(R$,LEN(S$)) THEN 200
130 PRINT R$;" COMES BEFORE ";S$
140 PRINT
150 GOTO 10
200 PRINT S$;" COMES BEFORE ";R$
210 PRINT
220 GOTO 10
230 END

RUN
NAME 1=? ADAMS
NAME 2=? BRONSON
ADAMS COMES BEFORE BRONSON

NAME 1=? SMITHY
NAME 2=? SMITH
SMITH COMES BEFORE SMITHY
```

Table 8. The program at the top shows the effects of the quirk in SWTPC 8K BASIC's test for string inequality. "SMITHY" comes before "SMITH." One possible fix appears in the program at the bottom. Now names will be put into conventional alphabetical order. (Underline = operator input.)

Tiny BASIC requires the least amount of memory, with Micro BASIC Plus a close second.

Here's the overall picture, then. Any one of the four might be best, given your specific circumstances . . . well, on second thought, I can't imagine any credible circumstances in which Sphere would come out on top. If you have just 4K of memory, you want Tiny BASIC. If you have 12 or 16K of RAM in your SWTPC 6800 and you want to have a full version with strings, you probably already have SWTPC 8K BASIC. If you have a 6800-based system, which leaves the bottom page of memory alone, you'll want to look into Micro BASIC Plus. If you don't have a system that uses the Motorola MIKBUG monitor system, and you feel apprehensive about grunging around in machine language trying to bring BASIC up, you probably want Tiny BASIC.

And if you insist on having a super-fast, super-cheap, easy-to-install version that will accept any ANSI Minimal BASIC program, well, you'll have to wait. ■

SELECTRIC TERMINAL

- 14.8 cps print speed
- 13 inch line length
- 10 or 12 characters/inch
- 6 or 8 lines/inch
- IBM 2740-1, 2 line control
- IBM 2740 keyboard/printer

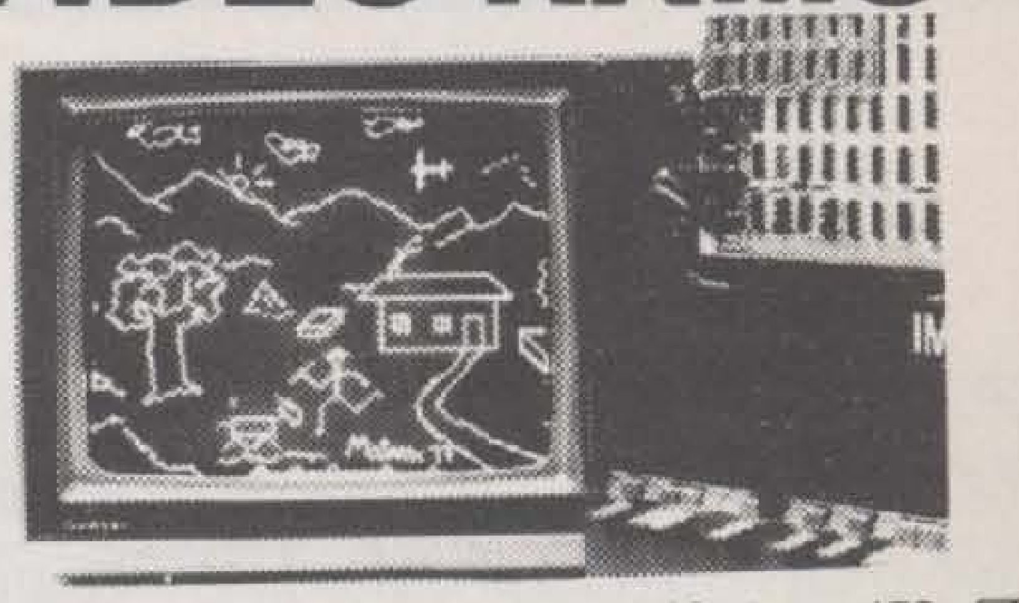
This deal can't be beat. Not only do you get a quality terminal for a fraction of the cost of anything else on the market, but you also get typewriter quality. Can be interfaced to a mini- or microcomputer or used with IBM computers. These terminals are worth the price of a typewriter alone. Our warranty is limited to replacing any defective parts for a period of 90 days following receipt of equipment. Interchangeable print element, of course. Includes attractive desk. Optional documentation package $25.00.

SUPER SALE

JUST **$675**

CALL (313) 994-3200 TO RESERVE

MATROX VIDEO RAMS



NCE/Compumart offers you the leader in CRT display controllers. Matrox Video Rams are complete with memory and require no processor time for screen refresh. Whether your application is alphanumeric, graphic or both, Matrox can provide a system to meet your needs.

MTX 816 Module (Upper Case only) Composite video output in 8 lines of 16 chars.	179	☐
MTX 1632 Module (Upper and Lower Case) 16 x 32 display includes blink option	225	☐
MTX 1632SL Module (Great with 2562)** Slave sync allows text superposition	225	☐
MTX 2480 Circuit Board 24 x 80 requires high-persistence phosphor (P39)	395	☐
MTX 2562 Circuit Board** 256 x 256 array graphic (multiples provide color)	630	☐
Matrox ALT-2480 Alpha-Numeric VRAM S-100, 24 lines of 80 char. (needs P39)	295	☐
Matrox ALT-2562 S-100,** Video Graphics Board Displays 256 x 256 array	395	☐

SORENSEN POWER SUPPLY

$31.50 SINGLE P.S. ☐ **$60.00** DOUBLE P.S. ☐

• precise dc output • 12v dc (nominal), 2 amp, 25 watt • 9-18 extended range • highly regulated-1.2 mv • accepts from 105-125v • accepts from 50-400 Hz, single phase • very low output ripple

This super quality power supply, from Raytheon, uses silicon transistors which give the unit a wide ambient-temperature operating range. Also, as a maintenance feature, they reduce the number of components. An excellent instruction manual is provided which includes schematics, parts list and troubleshooting information.

Name ______________________________

Address ______________________________

City State Zip ______________________________

Charge my BAC/VISA____ M/C____ Interbank # ______

Card # ____________________ Exp. Date ______

Signature ______________________________

☐ Please send me info on the new Commodore PET 2001 computer

Total for goods checked above ______

Please add 4% for Shipping/Handling + 4% tax if Mich. resident. ______

Total Amount ______

Limited 90-day Warranty
Return non-working merchandise within 90 days for replacement or refund. Kits are warranted to be complete with working components. All items subject to prior sale.

C41

NCE/CompuMart

1250 North Main Street, Department KB18
P.O. Box 8610 Ann Arbor, Michigan 48107
994-4445