In this article, Steve Wozniak describes how he calculated e to 47K of precision on an Apple II with only 48K of RAM. To achieve this, he had to remove almost all system software. Even video memory was used to store e, which was written to tape after calculation to preserve its value so that the Apple II monitor could print to the screen. (1981:6 p392)

The Impossible Dream:

Computing *e* to 116,000 Places with a Personal Computer

Stephen Wozniak
Apple Computer Inc
10260 Bandley Dr
Cupertino CA 95014

The 1960s were a decade of unrest, turbulence, and accomplishment. Man walked on the moon, *Star Trek* was launched, and the first million digits of π were determined by a computer. Today, as we face the early 1980s, Robert Truax, a backyard hobbyist, is constructing a private spacecraft, *Star Trek* has been revived as a movie, and personal computers are a reality. As a people, passion drives us to explore the unknown reaches of our universe. It is pleasing to note that this exploration is no longer the exclusive domain of governments and large institutions.

The purpose of this article is to share my experiences in computing the mathematical constant *e* to 116,000 digits of precision on an Apple II computer. Although this computation has little intrinsic value or use, the experience was stimulating and educational. The problems I was forced to overcome gave me insights that greatly contributed to new floating-point routines. These routines were, in some cases, two to three times as fast as those currently implemented in some of our languages at Apple. Because I wanted to develop my own solutions to the problem, I did not research existing techniques for computing *e* to great precision. Therefore, my approaches are quite possibly not state-of-the-art.

I first calculated *e* to 47 K bytes of precision in January 1978. The program ran for 4.5 days, and the binary result was saved on cassette tape. Because I had no way of detecting lost-bit errors on the Apple (16 K-byte dynamic memory circuits were new items back then), a second result, matching the first, was required. Only then would I have enough confidence in the binary result to print it in decimal.

Before I could rerun the 4.5 day program successfully, other projects at Apple, principally the floppy-disk controller, forced me to deposit the project in the bottom drawer. This article, already begun, was postponed along with it. Two years later, in March 1980, I pulled the *e* project out of the drawer and reran it, obtaining the same results. As usual (for some of us), writing the magazine article consumed more time than that spent meeting the technical challenges.

*Just before this issue went to press, Steve Wozniak told me that he had redesigned the theoretical "e-machine" that uses dedicated hardware for calculating e. The machine, which costs under $10,000, would use disk storage on a hard disk to replace large amounts of programmable memory. Steve estimates that a calculation of e to 100,000,000 places (ten times as many places as the current calculation of e) could be made in three months of calculation time....***GW**

Little Things Add Up

To compute the value of *e*, a method or formula must be found or derived. The *CRC Standard Mathematical Tables* handbook (see references) provides the well-known formula:

$$e = 1 + 1/1! + 1/2! + 1/3! + \ldots$$

We know that *e* is approximately 2.71828. For the sake of simplicity, we will deal with the fractional part only (.71828, etc) and abbreviate it *efrac*.

$$efrac = 1/2! + 1/3! + 1/4! + \ldots$$

Because each term is less than one-half the prior term, this series converges with the property that the sum of all terms beyond a specified *n*th term is less than that *n*th term. Thus, if the series is truncated after *n* terms, the maximum error in the computation is less than $(1/n!)$. This property relates the number of terms used, *n*, to the precision obtained in the computation. Because this series contains a factorial in the denominator of the terms, it is said to converge rapidly. This means that great precision can be obtained with relatively few terms. For example,

the *CRC Standard Mathematical Tables* handbook lists 100! as 9.3326×10^{157}, signifying that 100 terms will yield almost 158 digits of precision. The rate of convergence is sufficient that, for the problem at hand, neither algebraic manipulation of the series for faster convergence nor selection of a different formula is necessary.

Divide and Conquer

The following algorithm accomplishes the evaluation of the series for *e*. Of course, all critical routines should be implemented in highly optimized machine (assembly) language for speed. An extra hour spent optimizing the innermost loops could save days of computation time. Even self-modifying code should be used to save a critical microsecond! Binary arithmetic should be used to obtain maximal precision and the fastest possible computation time. Later, the result can be converted to decimal as it is printed.

The algorithm is as follows (also see figure 1):

1. Divide available memory equally into two arrays, TERM and E. The TERM array will contain successive terms (1/*i*!) and is initialized to 0.5 (1/2!). The E array will contain the running total of the terms and is also initialized to 0.5. Both arrays can be thought of as long bit streams of the fractional parts of the numbers they represent.
2. Set the variable DIVISOR to an initial value of 3.
3. Divide TERM by DIVISOR, forming 1/(DIVISOR!). Multiprecision division techniques will be discussed later.
4. Add TERM to E, keeping the assumed decimal points aligned. This sum will always be purely fractional (ie: it will never equal or exceed 1).
5. Increment the DIVISOR variable.
6. Repeat steps 3, 4, and 5 until TERM is reduced to all zeros or until a predetermined maximum divisor is reached.

This basic computation algorithm utilizes only 50% of available memory for the result. By rearranging the series for *e*, we can arrive at an approach that utilizes 100% of the memory.

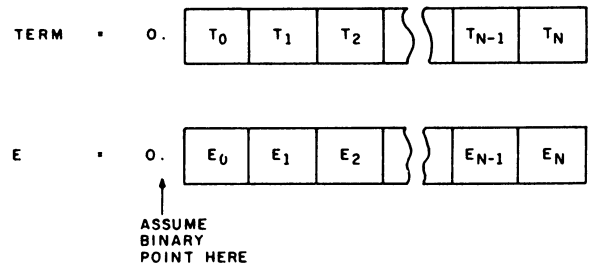

Figure 1: *Memory usage in the first algorithm to calculate e. Equal amounts of memory are devoted to a sequence of bytes representing the value of the current term being calculated (TERM) and the sum of all terms calculated thus far (E). Both numbers are seen as binary fractions (ie: the leftmost bit represents ½, the next bit represents ¼, etc).*

We begin by reversing the order of terms in *efrac*:

$$efrac = 1/2! + 1/3! + \ldots + 1/(n-1)! + 1/n! \ (n \text{ terms})$$
$$= 1/n! + 1/(n-1)! + \ldots + 1/3! + 1/2!$$

We then develop the following identity:

$$\frac{1}{i!} + \frac{1}{(i-1)!} = \frac{1}{i(i-1)!} + \frac{1}{(i-1)!}$$

$$= \frac{\frac{1}{i} + 1}{(i-1)!}$$

By repeatedly applying this identity to the formula, we get:

$$efrac = \cfrac{\cfrac{\cfrac{\cfrac{\frac{1}{n} + 1}{(n-1)} + 1}{\ddots}}{3} + 1}{2}$$

On inspection, the second series is equivalent to the first for *n* terms. A notable property of the new series is that the computation begins with the *n*th (greatest) divisor and ends with 2 (the smallest). The algorithm for computing *e* with this series is as follows:

1. Allocate all available memory to the E array (which stores the value of *efrac*, the fractional part of *e*). Initialize it to zero.
2. Set the initial value of DIVISOR to *n*, the precalculated maximum term (where *n*! is greater than the precision of the result to be computed).
3. Add 1 to E and divide by the current DIVISOR. The addition may simply imply setting the carry before dividing.
4. Decrement the DIVISOR.
5. Repeat steps 3 and 4 until the divisor equals 1.

Divisor	E (after step 3)
5	1/5
4	1/4 + 1/(4 × 5)
3	1/3 + 1/(3 × 4) + 1/(3 × 4 × 5)
2	1/2 + 1/(2 × 3) + 1/(2 × 3 × 4) + 1/(2 × 3 × 4 × 5)
	(1/2! + 1/3! + 1/4! + 1/5!)

Table 1: *Example of the calculation of e by the first algorithm.*

An example of this algorithm for $n=5$ is given in table 1.

How Large Is It?

An associate of mine once discovered that integrated circuit layouts could be conveniently specified in nanoacres! In the computation of e, it is more meaningful to specify the precision of the result in decimal digits rather than in the number of bytes allocated. The following formula performs the conversion:

$$\log_{10}(x) = \log_{256}(x) \times \log_{10}(256)$$
$$\text{(number of digits)} = \text{(number of bytes)} \times (2.40824)$$

For example, assume that 14 K bytes of memory are allocated to the fraction of e. The number of digits of accuracy this represents is given by the following:

$$\text{number of digits} = 14 \times 1024 \times 2.40824$$
$$= 34524.5 \text{ digits}$$

The process of calculating the number of terms needed to compute e to this precision is less straightforward. What must be determined is the minimum value of n, where $n!$ is greater than the precision corresponding to available memory. For the above example, this is the minimum n such that $n!$ is greater than 10^{34524}. The *CRC Standard Mathematical Tables* handbook lists Stirling's Formula, an equation useful for calculating the magnitude of $n!$ for reasonably large n:

$$\lim_{n\to\infty} \frac{n!\,\exp(n)}{n^{(n+0.5)}} = \sqrt{2\pi}$$

Taking the natural logarithms of both sides, we get:

$$\lim_{n\to\infty} \ln(n!) = \frac{\ln(2\pi)}{2} + [\ln(n)]\,[n+0.5] - n$$

Dividing by ln(10) to obtain the result in common (base-10) logarithms, we see the following:

$$\lim_{n\to\infty} \log_{10}(n!) = \frac{\log_{10}(2\pi)}{2} + [\log_{10}10(n)][n+0.5] - \frac{n}{\ln(10)}$$

The integer portion of this result gives us one less than the number of digits in $(n!)$.

The HP-41C calculator program in listing 1 calculates $\log_{10}(n!)$ (the number of digits in $n!$), given n.

By trial and error, it is easy to zero in on the minimum n for which $\log_{10}(n!)$ is greater than 34,524, the number of digits of precision corresponding to 14 K bytes of memory. Table 2 shows a set of values for n in the order in which they were calculated to find the desired value.

The value 9716 is found to be the minimum suitable value of n. Because it is difficult to relate the precision of $n!$ to that of $1/n!$, a slightly higher value (perhaps 9720) should be used for n. This will also compensate for minor formula or calculation errors.

A Multiprecision Division Algorithm

The problem at hand calls for the division of a very large dividend (possibly several kilobytes) by a moderate divisor (2 bytes). The general approach is to shift the divisor relative to the dividend, from the most significant bits toward the least, performing the familiar subtract/replace and shift technique that we call long division.

A few general optimizations should be considered. First, the following algorithm assumes that the divisor is less than 32,768 (2^{15}). If the divisor were to exceed 32,768, it would have to be compared to a value that could exceed 16 bits (2 bytes). Because indexed operations on the 6502 microprocessor are slower than absolute, direct, zero-page, or register operations, a few "fast" memory locations are allocated to hold the temporary (ie: relating to the current byte) dividend/quotient, and remainder. These locations are designated A0 (dividend/quotient), and A1 and A2 (2-byte remainder), and they should be allocated to the most accessible memory locations (or registers). The high-order byte of the fraction array E is assumed to be E(0), and the low-order byte is E(n). Remember that the 2-byte divisor, NH and NL, represents a whole number, and that the dividend represents a binary fraction with the binary point directly to the left of the MSB (most significant bit) of E(0).

In the algorithm that follows, the A0 byte represents the current byte, E(i), of the dividend at step 2. By step 6, however, all the digits of the dividend have been shifted out to the left (to the A1, A2 combination), and the digits of the new quotient have been shifted into A0 from the right. A0 is actually doing the work of two 8-bit registers.

Of course, all computation should be done in binary for maximum precision and speed. While targeted for 8-bit machines, these techniques are applicable to machines of longer word lengths.

The "add 1 and divide by n" algorithm (see figure 2) is as follows:

1. Initialize the remainder (locations A2 and A1) to 1, effectively adding 1.0 to the fractional dividend prior to dividing. (A2 is the most significant byte of the remainder.) This accommodates the algorithm developed for calculating e. An unmodified divide operation would call for initializing the remainder to zero. Initialize the index, i, to zero.
2. Move the next dividend byte, E(i), to location A0 to divide it by n. Shift A0 left 1 bit, moving the MSB into the carry bit.

Listing 1: *The FACTLOG program for the Hewlett-Packard HP-41C calculator. This program calculates the approximate number of digits in the number (n!).*

```
LBL ALPHA FACTLOG ALPHA ENTER LOG LASTX .5 + *
x<>y 10 LN / -
PI ENTER + LOG 2 / + RTN
```

3. Rotate the 16-bit remainder (A2 and A1) to the left by 1 bit, and rotate the carry bit from A0 into the LSB (least significant bit) of A1. This corresponds to the "shift" portion of the subtract-and-shift algorithm for division. No overflow can occur from this shift because the residual remainder must be less than twice the divisor, which in turn is less than 32,768 (2^{15}).
4. Compare the remainder, A2 and A1, to the divisor locations NH and NL. If the remainder is greater, then replace it with the difference of the two and set the quotient bit to 1. Otherwise, clear the quotient bit.
5. Rotate the quotient bit into the LSB of A0, and rotate the MSB of A0 into the carry bit.
6. Perform steps 3, 4, and 5, a total of eight times. Then replace E(*i*) with the byte in A0 (which is now the quotient of the byte-wide division just finished). Increment the index, *i*, and continue at step 2 until the last byte, E(*n*), has been processed.

n	$\log_{10}(n!)$ (number of digits in *n*!)
10000	35659.5
9000	31681.9
9700	34461.4
9800	34860.3
9730	34581.0
9720	34541.2
9710	34501.3
9715	34521.2
9716	34525.2

Table 2: *Trial-and-error determination of the number of terms, n, needed to obtain 34,524 digits of precision in the calculation of e. In the algorithm used to calculate e, the smallest contribution to the final value is made by the term (1/n!). The number of digits in (n!) is determined by estimating the value of n! and taking the logarithm to the base 10. The desired value of n is the first integer value greater than 34,524.*

Special Optimizations

I drive a small car and have found that it is helpful to accelerate or decelerate slightly in advance of certain stretches of the road (especially hills and downgrades) to obtain an adequate performance. Similarly, it is sometimes necessary to compensate for the inherent deficiencies of microprocessors (eg: their size) by carefully implementing specific optimizations. For example, the comparison performed in step 4 (discussed above) would normally be done by subtracting the low, and then high bytes, and possibly preserving the difference for replacement of the remainder. Within certain processors, it may be faster to first compare the high bytes, since they frequently dictate the comparison result (255 out of 256 times for arbitrary contents). Also, the critical steps 3, 4, and 5 can be coded eight times in-line to avoid the overhead time of a loop. And because the divisor changes infrequently, it can be coded as fast immediate-mode data. After each full divide, the code, which resides in programmable memory, can be modified for the next divisor.

The 6502 assembly-language program in listing 2 calculates *e* in 14 K bytes of memory. In order to keep the listing brief for this article, the program is not fully optimized. The major operation (add 1, divide) is not coded in-line eight times but is instead implemented as a loop. Because the Y register is used as a loop counter, it is not available as an index to the *e* array, and time-consuming increment instructions must be performed on the instructions at EREF1 and EREF2. Also, it is slightly faster not to move the current dividend byte of *e* into a separate fast location (A0 in the algorithm).

The *e* array begins at hexadecimal location 800 (which is the most significant byte of the array). This secondary text-screen page of the Apple II allows you to view

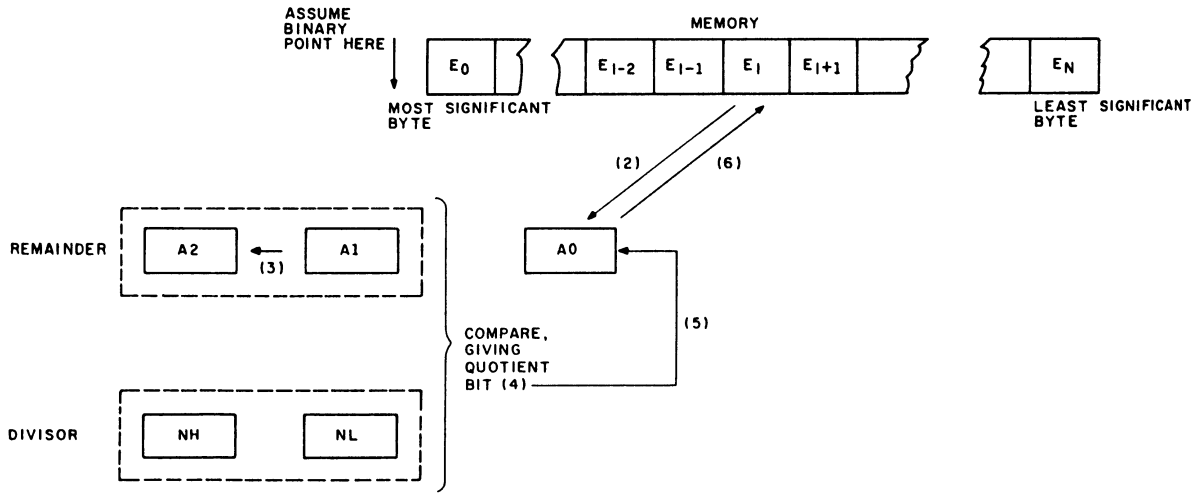

Figure 2: *Memory usage in the multiple-byte "add 1 and divide by n" division algorithm. The second algorithm (given in the text) reduces memory usage by 50% by using one long string of bytes in the computation process. The E array is divided 1 byte at a time by the 2-byte divisor. The A0 byte is used to store both the dividend and the quotient at different points in the algorithm. The numbers in parentheses refer to numbered steps in the algorithm.*

roughly the first 1 K bytes of *e* as they are calculated. Although the character representation is not readily useful, it is at least comforting to observe that the program is working on the correct section of memory. Do not execute this program until you read further and have a good idea of how long it runs before completion. Also, remember that although the result is in binary and somewhat meaningless, it will later be converted to decimal and printed.

Tomorrow Is a Long Time

The execution time of this program is proportional to the number of divisions performed (9719 for the above example), the number of bytes being divided (14 K bytes in this case), and the average divide time per byte.

The average divide time per byte is calculated as follows. In listing 2, the numbers in parentheses are the cycle times of all significant instructions of the divide routine. Careful analysis shows that when the high-order dividend (remainder) byte is less than the high-order divisor byte, 23 cycles are used. When the former is greater than or equal to the latter, 39 cycles are used, with approximately 13.5 additional cycles (on the average) if the two are equal. Statistically, the remainder will be less than the divisor half of the time and greater than or equal to the divisor half of the time. Analysis reveals that the 2 bytes will be equal approximately one out of every 2*H* comparisons, where *H* is the high-order divisor byte contents. In the example, *H* varies from 37 down to 0, so the average frequency of equality is 1 in 37. Using this "fudge factor," the average cycle time per 1-bit partial division is computed as follows:

$$\text{cycles per bit} = {}^{23}/_{2} + {}^{39}/_{2} + {}^{13.5}/_{37}$$
$$= 31.3649 \text{ cycles}$$

Every byte divided includes eight of the above iterations plus an overhead of 21 cycles, giving the following average:

$$\text{cycles per byte} = (\text{cycles per bit} \times 8 \text{ bits per byte}) + 21$$
$$= 31.3649 \times 8 + 21$$
$$= 271.919 \text{ cycles}$$

The average time per cycle on the Apple II is a function of the crystal frequency (14.31818 MHz) and the fre-

Listing 2: *A 6502 machine-language program for calculating e to 34,524 decimal digits. The result is in binary and must be converted to decimal by the programs shown in listings 3 and 4.*

```
SOURCE FILE: ECALC1
0000:             1          LSTON
0000:             2 ***********************************
0000:             3 *                                 *
0000:             4 *    CALCULATION OF E -- 14K      *
0000:             5 *                                 *
0000:             6 *       WOZ       20-APR-80       *
0000:             7 *                                 *
0000:             8 *         EXAMPLE PROGRAM         *
0000:             9 *                                 *
0000:            10 ***********************************
0000:            11 *                                 *
0000:            12 *  LOCATIONS $800-3FFF ARE USED   *
0000:            13 *  FOR THE (BINARY) FRACTION OF   *
0000:            14 *  E. LOCATION $800 IS THE MOST   *
0000:            15 *  SIGNIFICANT BYTE, $3FFF IS     *
0000:            16 *  THE LEAST SIGNIFICANT.  THIS   *
0000:            17 *  CORRESPONDS TO APPROXIMATELY   *
0000:            18 *  34524 DIGITS.                  *
0000:            19 *                                 *
0000:            20 ***********************************
0000:            21 *                                 *
0000:            22 *  THE FIRST DIVISOR IS 9720      *
0000:            23 *  AND THE LAST IS 2.  9720       *
0000:            24 *  FACTORIAL IS GREATER THAN      *
0000:            25 *  10 ^ 34524.                    *
0000:            26 *                                 *
0000:            27 ***********************************
0000:            28 *                                 *
0000:            29 *  THE MAJOR OPERATION IS AN      *
0000:            30 *  INCREMENT (+1) OF E FOLLOWED   *
0000:            31 *  BY A MULTI-PRECISION DIVIDE    *
```

```
0000:           32 *  BY THE CURRENT DIVISOR.      *
0000:           33 *  EACH SUCCESSIVELY LESS SIG-  *
0000:           34 *  NIFICANT BYTE OF E, TOGETHER *
0000:           35 *  WITH THE RESIDUAL REMAINDER  *
0000:           36 *  A1 AND A2, IS DIVIDED BY THE *
0000:           37 *  CURRENT 2-BYTE DIVISOR.  THE *
0000:           38 *  8-BIT QUOTIENT IS LEFT IN E  *
0000:           39 *  AND THE RESIDUAL REMAINDER   *
0000:           40 *  IN A1 AND A2 (ACC HOLDS A2). *
0000:           41 *                               *
0000:           42 *********************************
0000:           43 A1      EQU  0         (CURRENT BYTE OF E IS A0, ACC IS A2)
0001:           44 PCOUNT  EQU  1         COUNTS RAM PAGES OF E ARRAY.
0800:           45 E       EQU  $800      E, BINARY FRACTION, TO $3FFF.
0038:           46 NUMPAG  EQU  $38       14K IS 56 RAM PAGES.
25F8:           47 N       EQU  9720      (N FACTORIAL IS > 34524 DIGITS)
25F8:           48 NL      EQU  N&$FF     LO BYTE OF N.
0025:           49 NH      EQU  N/256     HI BYTE OF N.
----- NEXT OBJECT FILE NAME IS ECALC1.OBJ0
0240:           51         ORG  $240
0240:A9 38      52 NXTDVSR LDA  #NUMPAG   INIT RAM PAGE COUNTER
0242:85 01      53         STA  PCOUNT      FOR 56 PAGES.
0244:A9 01      54         LDA  #1
0246:85 00      55         STA  A1        INIT RESIDUAL REMAINDER TO 1. (FOR +1)
0248:A9 08      56         LDA  #E/256
024A:8D 5C 02   57         STA  EREF1+2   MODIFY CODE SO THAT REFS
024D:8D 78 02   58         STA  EREF2+2     TO E POINT TO FIRST BYTE.
0250:A9 00      59         LDA  #0        (ACC IS ALSO A2 OF RESIDUAL REMAINDER)
0252:8D 5B 02   60         STA  EREF1+1
0255:8D 77 02   61         STA  EREF2+1
0258:A0 08      62 NXTBYTE LDY  #8        (2) COUNTER--8 BITS PER BYTE.
025A:0E 00 08   63 EREF1   ASL  E         (6) MSB OF DIVIDEND BYTE TO CARRY.
025D:26 00      64 NXTBIT  ROL  A1        (5) SHIFT 3-BYTE DIVIDEND.
025F:2A         65         ROL  A         (2)  (ACC IS A2)
0260:C9 25      66 NHREF1  CMP  #NH       (2) IF HI BYTE LESS THAN DIVISOR
0262:90 12      67         BCC  EREF2     (3/2)   THEN QUOTIENT BIT IS 0.
0264:D0 06      68         BNE  REPLACE   (3/2) (TAKEN IF GREATER)
0266:A6 00      69         LDX  A1        (3) COMPARE LOW BYTES IF HI BYTES EQUAL.
0268:E0 F8      70 NLREF1  CPX  #NL       (2)
026A:90 0A      71         BCC  EREF2     (3/2) IF LESS, QUOTIENT BIT IS 0.
026C:AA         72 REPLACE TAX            (2)
026D:A5 00      73         LDA  A1        (3) REPLACE RESIDUAL REMAINDER A1 AND A2
026F:E9 F8      74 NLREF2  SBC  #NL       (2)   WITH RESIDUAL REMAINDER
0271:85 00      75         STA  A1        (3)   MINUS CURRENT DIVISOR.
0273:8A         76         TXA            (2) (HI BYTE OF RESIDUAL REMAINDER)
0274:E9 25      77 NHREF2  SBC  #NH       (2) (GUARANTEED TO SET CARRY)
0276:2E 00 08   78 EREF2   ROL  E         (6) QUOTIENT BIT INTO A0 LSB, MSB TO CARRY.
0279:88         79         DEY            (2) NEXT OF 8 BITS.
027A:D0 E1      80         BNE  NXTBIT    (3/2) LOOP--NOTE: CARRY = QUOTIENT BIT.
027C:EE 5B 02   81         INC  EREF1+1   (5)
027F:EE 77 02   82         INC  EREF2+1   (5) MODIFY CODE REFS TO E ARRAY.
0282:D0 D4      83         BNE  NXTBYTE   (3)  (NO BYTE OVERFLOW)
0284:EE 5C 02   84         INC  EREF1+2
0287:EE 78 02   85         INC  EREF2+2   (MODIFY HI BYTE)
028A:C6 01      86         DEC  PCOUNT
028C:D0 CA      87         BNE  NXTBYTE   LOOP UNTIL DONE 56 RAM PAGES.
028E:AD 69 02   88         LDA  NLREF1+1
0291:D0 06      89         BNE  NXTDVR2
```

```
0293:CE 61 02    90          DEC   NHREF1+1  DECR IMMEDIATE REFS TO
0296:CE 75 02    91          DEC   NHREF2+1    CURRENT DIVISOR.
0299:CE 69 02    92 NXTDVR2  DEC   NLREF1+1
029C:CE 70 02    93          DEC   NLREF2+1
029F:AD 69 02    94          LDA   NLREF1+1
02A2:4A          95          LSR   A
02A3:0D 61 02    96          ORA   NHREF1+1  LOOP IF DIVISOR > 1.
02A6:D0 98       97          BNE   NXTDVSR
02A8:60          98          RTS             (DONE)

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Listing 3: *A BASIC driver program to print e from binary to decimal form. The program uses the machine-language program EPRNT, shown in listing 4.*

```
SOURCE FILE: EPRNT
0000:             1 ********************************
0000:             2 *                              *
0000:             3 *    'E' PRINTOUT ROUTINES     *
0000:             4 *                              *
0000:             5 *          14K VERSION         *
0000:             6 *                              *
0000:             7 *      WOZ       20-APR-80     *
0000:             8 *                              *
0000:             9 ********************************
0000:            10 *                              *
0000:            11 *  THESE SUBROUTINES PERFORM   *
0000:            12 *  THE CRITICAL OPERATIONS     *
0000:            13 *  FOR CONVERTING THE 14K      *
0000:            14 *  BINARY VERSION OF 'E'       *
0000:            15 *  TO DECIMAL FOR PRINTING.    *
0000:            16 *  THEY ARE INTENDED TO BE     *
0000:            17 *  CALLED FROM A BASIC PROGRAM *
0000:            18 *  WHICH DOES THE ACTUAL       *
0000:            19 *  PRINTING.                   *
0000:            20 *                              *
0000:            21 ********************************
0000:            22 *                              *
0000:            23 *  THE BINARY REPRESENTATION   *
0000:            24 *  OF THE FRACTIONAL PART OF   *
0000:            25 *  E (OR ANY OTHER NUMBER      *
0000:            26 *  TO BE CONVERTED TO DECIMAL) *
0000:            27 *  IS STORED IN LOCATIONS $800 *
0000:            28 *  (MOST SIGNIFICANT) TO $3FFF *
0000:            29 *  (LEAST).  THE SUBROUTINES   *
0000:            30 *  INIT AND MULT RESIDE IN THE *
0000:            31 *  $4000 PAGE OF MEMORY AND    *
0000:            32 *  USE TABLES PRODLO AND       *
0000:            33 *  PRODHI IN THE $4100 AND     *
0000:            34 *  $4200 PAGES RESPECTIVELY.   *
0000:            35 *  LOMEM MUST BE SET TO $4300  *
0000:            36 *  (17152 DECIMAL) OR GREATER  *
0000:            37 *  FROM BASIC.                 *
0000:            38 *                              *
0000:            39 ********************************
0000:            40 *                              *
```

```
0000:            41 *  SUBROUTINE INIT MUST BE     *
0000:            42 *  CALLED ONCE TO GENERATE     *
0000:            43 *  'MULTIPLY BY 100' TABLES    *
0000:            44 *  PRODLO AND PRODHI.  INIT    *
0000:            45 *  MUST BE CALLED BEFORE MULT. *
0000:            46 *                              *
0000:            47 *  SUBROUTINE MULT PERFORMS    *
0000:            48 *  A 'MULTIPLY BY 100' ON THE  *
0000:            49 *  NUMBER 'E'.  IT RETURNS     *
0000:            50 *  THE NEXT TWO DIGITS OF THE  *
0000:            51 *  DECIMAL EQUIVALENT AS A     *
0000:            52 *  NUMBER BETWEEN 0 AND 99 IN  *
0000:            53 *  LOCATION 1 (WHERE BASIC     *
0000:            54 *  CAN PEEK IT FOR PRINTING).  *
0000:            55 *                              *
0000:            56 ********************************

0000:            58 XSAV    EQU  0          X-REG SAVE LOCATION.
0001:            59 RESULT  EQU  1          RESULT BYTE FROM MULTIPLY.
0002:            60 PCOUNT  EQU  2          COUNTS NUMBER OF RAM PAGES OF E.
4100:            61 PRODLO  EQU  $4100      LOW BYTE TABLE (100 * IDX).
4200:            62 PRODHI  EQU  $4200      HI BYTE TABLE (100 * IDX).
0800:            63 E       EQU  $800       E, BINARY FRACTION, TO $3FFF.
0038:            64 NUMPAG  EQU  56         56 PAGES IN 14K
003F:            65 LASTPAG EQU  $3F        LAST (LEAST SIGNIFICANT) PAGE OF E.
0000:            66 *
0000:            67 ************************************************************
0000:            68 *
----- NEXT OBJECT FILE NAME IS EPRNT.OBJ0
4000:            69         ORG  $4000
4000:86 00       70 INIT    STX  XSAV       PRESERVE X-REG FOR INT BASIC.
4002:A9 00       71         LDA  #0         STARTING PRODUCT LO BYTE.
4004:AA          72         TAX             STARTING PRODUCT HI BYTE.
4005:A8          73         TAY             STARTING INDEX TO PRODUCT TABLES.
4006:99 00 41    74 PRODGEN STA  PRODLO,Y   STORE LOW BYTE OF 100 * Y.
4009:48          75         PHA             PRESERVE A-REG
400A:8A          76         TXA             HI BYTE OF CURRENT PRODUCT.
400B:99 00 42    77         STA  PRODHI,Y   STORE HI BYTE OF 100 * Y.
400E:68          78         PLA             RESTORE A-REG (PRODUCT LOW BYTE).
400F:18          79         CLC
4010:69 64       80         ADC  #100       ADD 100 FOR NEXT PRODUCT.
4012:90 01       81         BCC  NXTPROD
4014:E8          82         INX
4015:C8          83 NXTPROD INY             NEXT OF 256 PRODUCTS.
4016:D0 EE       84         BNE  PRODGEN
4018:A6 00       85         LDX  XSAV       RESTORE X-REG FOR INT BASIC.
401A:60          86         RTS             (RETURN
401B:            87 *
401B:            88 ************************************************************
401B:            89 *
401B:A9 38       90 MULT    LDA  #NUMPAG
401D:85 02       91         STA  PCOUNT     56 PAGES IN 14K.
401F:A9 3F       92         LDA  #LASTPAG
4021:8D 32 40    93         STA  MULT1+2    INIT E REFS FOR LEAST
4024:8D 38 40    94         STA  MULT2+2      SIGNIGICANT RAM PAGE.
```

```
4027:A0 00      95          LDY  #0         INIT INDEX TO E (WILL DECR TO $FF FIRST TIME)
4029:A2 00      96          LDX  #0         TRICK TO CLEAR RESIDUAL CARRY.
402B:18         97          CLC
402C:BD 00 42   98 MULBYT   LDA  PRODHI,X   (4) HI PROD BYTE IS RESIDUAL CARRY.
402F:88         99          DEY             (2) NEXT MORE SIGNIFICANT BYTE OF E.
4030:BE 00 08  100 MULT1    LDX  E,Y        (4) (GET IT)
4033:7D 00 41  101          ADC  PRODLO,X   (4) TIMES 100, PLUS RESIDUAL CARRY.
4036:99 00 08  102 MULT2    STA  E,Y        (5) RESTORE PRODUCT BYTE.
4039:98        103          TYA             (2) LAST BYTE THIS PAGE?
403A:D0 F0     104          BNE  MULBYT     (3/2)  NO, CONTINUE.
403C:CE 32 40  105          DEC  MULT1+2    (6)
403F:CE 38 40  106          DEC  MULT2+2    (6) NEXT MORE SIGNIFICANT PAGE.
4042:C6 02     107          DEC  PCOUNT     (5) DONE 56 PAGES?
4044:D0 E6     108          BNE  MULBYT     (3)   NO, CONTINUE.
4046:7D 00 42  109          ADC  PRODHI,X   RETRIEVE FINAL CARRY.
4049:85 01     110          STA  RESULT     SAVE AS TWO-DIGIT RETURNED VALUE.
404B:A6 00     111          LDX  XSAV       RESTORE X-REG FOR INT BASIC.
404D:60        112          RTS             (RETURN)

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Listing 4: *EPRNT, a machine-language program that converts a binary number for printing as a decimal number.*

```
FORMATTER PROGRAM - APPLE INTEGER BASIC

FILE E1 IS 'E' FROM $800 TO $3FFF

FILE EPRNT.OBJ0 IS INIT AND MULT SUBRS

CAUTION: MUST SET LOMEM TO 17152!

  10 D$="": PRINT D$;"NOMON C,I,O": PRINT D$;"BLOAD E1,A$800": PRINT D$;
     "BLOAD EPRNT.OBJ0,A$4000": PRINT D$;"PR#2"
  20 INIT=16384:MULT=16411: CALL INIT:ODDEVEN=0
  30 FOR PAGE=1 TO 10: PRINT : PRINT "      E";: FOR I=1 TO 63: PRINT " "
     ;: NEXT I: PRINT "PAGE ";PAGE/10;PAGE MOD 10: PRINT
  40 FOR LINE=1 TO 60: IF PAGE>1 OR LINE>1 THEN 50: PRINT "  E=2.";: GOTO
     60
  50 PRINT "      ";
  60 FOR GROUP=1 TO 12
  70 FOR DIG=1 TO 5: GOSUB 200: NEXT DIG
  80 PRINT " ";: NEXT GROUP
  90 PRINT : IF PAGE=10 AND LINE=35 THEN 110: NEXT LINE: REM QUIT AFTER 34500
      DIGITS
 100 PRINT : PRINT : PRINT : NEXT PAGE
 110 PRINT D$;"PR#0": END : REM  TURN PRINTER OFF
 190 REM
 192 REM  SUBROTINE 200 PRINTS NEXT DIG
 194 REM
 200 IF ODDEVEN=1 THEN 220: CALL MULT
 210 PRINT PEEK (1)/10;: GOTO 230
 220 PRINT PEEK (1) MOD 10;
 230 ODDEVEN=1-ODDEVEN: RETURN
```

quency-dividing circuitry that generates the microprocessor clock. Due to color-graphics considerations, a slight adjustment (to eliminate display jitter) is made, which introduces a constant multiplying the crystal period, and gives us the following time per machine cycle:

$$\begin{aligned}\text{time per cycle} &= 912/((65)(14.31818\ \text{MHz})) \\ &= 0.9799269\ \mu s\end{aligned}$$

The division time per byte (in μs) and time per program execution can now be calculated:

$$\begin{aligned}\text{time per byte} &= \text{cycles per byte} \times \text{time per cycle} \\ &= 271.919\ \text{cycles} \times .9799269\ \mu s \\ &\quad \text{per cycle} \\ &= 266.46\ \mu s \\ \text{time per program} &= \text{time per byte} \times \text{number of} \\ &\quad \text{bytes} \times \text{number of divisions} \\ &= 266.46\ \mu s \times (14)(1024) \times 9719 \\ &= 37{,}126\ \text{seconds} \\ &= 10.3\ \text{hours}\end{aligned}$$

Note that as you compute *e* to greater precision, both the number of divisors and the length of each division increase. Also, at some point, a 2-byte division no longer suffices and a 3-byte division must be used. This causes the execution time to vary with roughly the second power of the precision sought. For example, three times the precision takes ten times as long to calculate!

Running the Example Program

If you wish to try the example program before branching out on your own, a few suggestions should be heeded. First, it is a shame to run a program for 10 hours and then find out it contained a minor bug. By changing N (the maximum divisor) to 1000 and NUMPAG to 4 (for 1 K bytes of precision), a quick trial/practice version can be assembled. The practice run allows the user to get the obvious mistakes out of the way with minimum consequence and verify that the assembly is correct. The following commands will clear the memory locations used, run the program, and finish in about 4.5 minutes (273 seconds). Hexadecimal location 0800 should contain B7, and location 0BFF should contain 24 upon completion. As mentioned previously, you can watch the calculation proceed by displaying the secondary text screen on the Apple II. During the trial run, it should be constantly changing.

The following two lines (to be entered when the Apple II is in monitor mode) allow you to run the test program:

```
*800:0 N801<800.BFEM
*C055 240G C054
```

The first line clears the area of memory that will be used, and the second line switches the video display to text page 2 (which will contain the value of *e* being computed), runs the program of listing 2, then returns to text page 1 when the program is complete.

The real (10-hour) example program should be run twice, and the results compared to verify that the program does not contain a minor bug and that the constants were properly determined. As discussed below, it is not necessary to initialize memory before running the program if the constant *n* has been properly selected. Therefore, it is recommended that the program be run first with initialized memory and later with random (uninitialized) memory. These results, when compared, should be identical. Once you have confidence in the binary result, save it on tape or floppy disk for printing in decimal.

Go Forth and Multiply

The computed binary fraction must next be converted to decimal and printed. The general method of converting a binary fraction to a decimal fraction is to repeatedly multiply it by decimal 10 (in binary). The carry from each multiplication (integer portion of product) is the next decimal digit. Because the most significant digits are generated first, the result can be printed as it is generated.

A higher-level language such as BASIC should be used to format the output, but unless you are planning a short vacation, highly optimized machine language should be used for the base conversion. The 6502 programs in listing 3 accomplish the conversion. Subroutine INIT is called once to generate a 256-entry, multiply-by-100 lookup table. Subroutine MULT scans the *e* array, from the least toward the most significant bytes, multiplying each byte by 100 via a fast table lookup. It also handles carries. The resultant carry is a 2-digit number between 0 and 99 that is returned to BASIC for printing. Note that multiplying by 100, instead of 10, generates 2 digits per pass.

Seeing Is Believing

The BASIC formatting program in listing 4 should produce an attractive printout. No single program will suffice, due to the fact that printers and people are so varied. The considerations include page headers (title, date, page number), lines per page, spacing between lines, digits per line, digit groupings (eg: groups separated by a space or two), and margins. For example, the poor horizontal registration of a Centronics 779 printer is painfully obvious with single-spaced printouts but almost undetectable with double-spaced ones. A little trial and error will insure that your printout is a perfect "10."

The program in listing 4 was used with an NEC (Nippon Electric Company) Spinwriter. It prints 60 digits per line (twelve groups of 5 digits, separated by single blanks) and 60 lines per page. The page heading is simply the letter *e* and the page number, carefully aligned with the left and right margins. The text "e=2." precedes the first digit of the printout. The program ends after printing 34,500 digits, despite the fact that an additional 24 digits are re-

quired in order to be correct. The final page and line number were precalculated to detect this stopping point. Lines 200 thru 230 make up a digit-printing subroutine that calls the assembly-language multiply-by-100 routine (MULT) every other digit.

Analysis of the Algorithm

The specified algorithm has the property that the contents of *e* at a given stage of computation will yet be divided by (*i*!), where *i* is the current divisor. The first implication of this property is that the allocated memory need not be initialized, since it will all be reduced to insignificance when divided by *n*! (because *n*, the starting divisor, was specifically chosen such that *n*! is greater than the significance corresponding to that much memory).

An interesting aspect of this implication is that the result is perfect to the last calculated bit, despite the fact that terms beyond the *n*th have been omitted. Additional terms (before the *n*th) would simply cause the allocated memory to have different contents (ie: be initialized arbitrarily) when the *n*th term is reached. Since division proceeds from high toward low significant bits, arbitrary data beyond a specified least significant byte can never affect the contents of that byte or any more significant byte. There can be no accumulated truncation errors such as those encountered with summation-of-terms approaches.

The second implication is that, at a given stage of calculation, only the most significant bytes of *e* (ie: those that will not subsequently be divided to insignificance) need to be divided! The first divisions can be very short, only a few bytes or so, while the last ones must encompass all of *e*. For a given divisor, *i*, the number of (least significant) bytes of *e* which need *not* be divided is $\log_{256}(i!)$, which may be calculated by the HP-41C program in listing 5. Note that it calls the previously written program FACTLOG, which calculates the number of digits of (*i*!). The algorithm used is:

number of bytes of $i!$ = number of digits of $i!/\log_{10}(256)$

It is unfeasible to precalculate the number of bytes to leave undivided (or the number to divide) for each divisor and to save it in a table because the table would consume a great deal of memory. As an alternative, the divisors can be broken into blocks of, say, 1 K bytes each, and for each block a fixed number of bytes (of *e*) grouped into fifteen blocks of 2 K-byte divisors each, and the number of memory pages not to be divided were precalculated for each block (see table 3). This version of the program used a lookup table to determine how many pages to divide (188 minus the number *not* to divide) for each divisor. This technique proved extremely beneficial because it reduced the computation time from four days to two.

Range of Divisors in Same Group	Number of Insignificant Bytes	Number of Pages That Can Be Left Uncalculated
2 to 2047	0	0
2048 to 4905	2448	9.6
4096 to 6143	5406	21.1
6144 to 8191	8558	33.4
8192 to 10239	11836	46.2
10240 to 12287	15206	59.4
12288 to 14335	18652	72.9
14336 to 16383	22158	86.6
16384 to 18431	25718	100.5
18432 to 20479	29325	114.5
20480 to 22527	32972	128.8
22528 to 24575	36656	143.2
24576 to 26623	40374	157.7
26624 to 28671	44123	172.4
28672 to 30719	47900	187.1

Table 3: *Table of truncated multibyte divisions that can be made during the second algorithm. Due to the nature of the second algorithm, most divisors need not carry the division out the entire length of the multibyte dividend. By grouping divisors and not calculating the bytes that are unimportant to that particular group, calculation time can be significantly decreased.*

The 47 K-byte version used virtually all the memory in a 48 K-byte Apple. The *e* array occupied hexadecimal locations 400 thru BFFF. A starting divisor of 28,800 can be divided. The number of bytes to divide for a given block is calculated as the total number of bytes in the *e* array minus the number of insignificant bytes (calculated as above) corresponding to the minimum divisor of the block, plus a "guard" byte or two to cover slight calculation errors.

In a later program that calculated *e* to 116,000 digits, I used 47 K bytes (188 pages of 256 bytes each) of memory, and the maximum divisor was 28,800. The divisors were resulted in 115,925 digits of precision. Because the result occupied screen memory, it had to be written to cassette tape by the calculation program before returning to the Apple II monitor. Because there was no memory available for a BASIC program, the output formatting program was coded in assembly language and resided in parts of pages 0 and 1. Pages 2 and 3 were used for the multiply-by-100 tables.

On the Horizon

As with any limitless search, there remains the challenge to compute *e* to even greater precision. Unfortunately, the computation time of the specified algorithm is exponentially related to the precision sought. Divide operations on high-speed computers (approximately 12

Listing 5: *The FACTBYT program for the Hewlett-Packard HP-41C calculator. This program calculates the precision to which the multibyte division has to be carried out for a given divisor. See table 3 for details.*

```
LBL ALPHA FACTBYT ALPHA XEQ ALPHA FACTLOG ALPHA
256 LOG / RTN
```

Listing 6: *A partial printout of the value of e. The first line agrees with the fifty-place value for e that is given in the* CRC Standard Mathematical Tables.

```
E                                                                    PAGE 01

E=2.71828 18284 59045 23536 02874 71352 66249 77572 47093 69995 95749 66967
  62772 40766 30353 54759 45713 82178 52516 64274 27466 39193 20030 59921
  81741 35966 29043 57290 03342 95260 59563 07381 32328 62794 34907 63233
  82988 07531 95251 01901 15738 34187 93070 21540 89149 93488 41675 09244
  76146 06680 82264 80016 84774 11853 74234 54424 37107 53907 77449 92069
  55170 27618 38606 26133 13845 83000 75204 49338 26560 29760 67371 13200
  70932 87091 27443 74704 72306 96977 20931 01416 92836 81902 55151 08657
  46377 21112 52389 78442 50569 53696 77078 54499 69967 94686 44549 05987
  93163 68892 30098 79312 77361 78215 42499 92295 76351 48220 82698 95193
  66803 31825 28869 39849 64651 05820 93923 98294 88793 32036 25094 43117
  30123 81970 68416 14039 70198 37679 32068 32823 76464 80429 53118 02328
  78250 98194 55815 30175 67173 61332 06981 12509 96181 88159 30416 90351
  59888 85193 45807 27386 67385 89422 87922 84998 92086 80582 57492 79610
  48419 84443 63463 24496 84875 60233 62482 70419 78623 20900 21609 90235
  30436 99418 49146 31409 34317 38143 64054 62531 52096 18369 08887 07016
  76839 64243 78140 59271 45635 49061 30310 72085 10383 75051 01157 47704
  17189 86106 87396 96552 12671 54688 95703 50354 02123 40784 98193 34321
  06817 01210 05627 88023 51930 33224 74501 58539 04730 41995 77770 93503
  66041 69973 29725 08868 76966 40355 57071 62268 44716 25607 98826 51787
  13419 51246 65201 03059 21236 67719 43252 78675 39855 89448 96970 96409
  75459 18569 56380 23637 01621 12047 74272 28364 89613 42251 64450 78182
  44235 29486 36372 14174 02388 93441 24796 35743 70263 75529 44483 37998
  01612 54922 78509 25778 25620 92622 64832 62779 33386 56648 16277 25164
  01910 59004 91644 99828 93150 56604 72580 27786 31864 15519 56532 44258
  69829 46959 30801 91529 87211 72556 34754 63964 47910 14590 40905 86298
[illegible]
E                                                                    PAGE 10

  92105 78191 37103 01889 79206 40888 39747 67667 14472 73142 54467 92350
  05246 18849 23745 53075 75734 90270 73424 96298 87999 69420 94595 96100
  87025 01329 45332 53580 45689 28570 72412 07965 91980 92255 50560 06197
  12835 41270 20207 25839 94171 17552 09208 20151 09650 95266 85113 89757
  71508 10849 44350 82854 58749 91294 38575 63115 66832 45668 27992 99186
  15390 09255 87171 68404 95663 99195 91540 34218 36453 72120 23678 60865
  53647 45175 65487 93189 25644 08527 44891 90918 19341 16675 83563 43975
  88860 46349 41311 18752 41038 42546 79379 99203 54691 04119 35443 11321
  91360 68129 65756 85836 11774 56465 46748 61061 98859 14148 05799 31872
  53675 31243 47033 54826 37527 08135 31055 70818 04964 24985 84646 14797
  34675 99315 94651 47870 25065 27108 35087 82350 65653 23317 97738 65666
  61816 52390 01766 49884 85456 05496 13002 15776 11525 58133 96184 02706
  78149 00350 25287 68236 07822 10739 71023 39146 87015 97358 68589 01529
  70103 47780 50329 21540 14359 59529 86834 04657 47175 62321 96640 51540
  14779 53167 46172 62087 27304 82063 46524 69109 95332 73755 61090 57837
  84559 45469 16022 36876 89641 42596 01646 89647 10634 80741 09928 54648
  23530 83540 13233 29248 64037 31800 31952 02317 47620 65377 26163 71744
  53605 49726 69060 17111 76761 04777 49716 66890 15216 38389 74311 71418
  06222 22345 71856 79415 07299 52620 10862 05084 78312 74747 91909 99688
  99372 75229 05367 47850 20500 03863 00365 26218 80067 09266 74104 80602
  73419 97756 66002 94279 41090 40006 46542 81074 45400 76164 29525 36246
  02614 76180 47174 43228 89953 28582 83977 62184 60096 76692 67581 27030
  28065 19535 45205 31735 36808 95458 99021 80783 14577 58912 80203 97005
  36331 93821 10009 54432 41244 19794 91929 16205 23442 13463 95653 84078
  12094 16214 83500 11558 83618 42116 42839 92454 02759 07196 21537 57018
  70670 83731 01224 61413 62048 92655 56681 09467 07638 65360 83015 84761
[illegible]
```

μs per 32 bits) are two orders of magnitude faster than the 6502 routines. The ultimate approach is to construct a custom "divide machine." Current technologies and low programmable memory prices make it feasible to construct such a machine with a thousand-fold performance improvement over the 6502 microprocessor. With such a machine, *e* could be computed to 100,000,000 digits within a couple of years (one year constructing and testing, one year computing). Such a machine would require power supply backup and error-correcting memory. The memory should be purchased at the latest possible date due to decreasing prices.

Once a few simple concepts are understood, the computation that I have described is as easy as *pi* (see listing 6). Why do people spend time computing these numbers to such absurd precision? Because they're there, I suppose. Who knows what great discoveries will be made by personal computer owners in the coming years? Rest assured that a guaranteed place in the mathematics Hall of Fame awaits the discoverer of the next greatest prime number. ■

What new form of democracy is required? Ours is 200 years old and was designed when it took weeks and even months for information to move across the country. How should we govern ourselves in a world where our president can ask the American people, say, their take on an issue, and then get their accumulated answers live, for all to see, as he talks to them on TV? — Robert Metcalfe, Principal Inventor of Ethernet and Founder of 3Com (1991:11 p119)

Windows runs sluggishly on any machine slower than a PC AT with a 20-megabyte hard disk drive — David and Lee Hart, BYTE Authors (1987:6 p250)

Readers will recall that I am no enthusiast of the key layout on the IBM Personal Computer (PC). The company has put extra keys between the normal typewriter-key layout's Z key and the Shift key, and it has reduced the size of the Return key and moved it far, far away from the home keys. It's an understatement to say I'm no enthusiast: indeed, I think it is (1) an insult to American touch-typists and (2) an unmitigated disaster. (I'm reminded of the lawyer who sent a telegram saying, "Sir: F— You. Strong letter follows.") — Jerry Pournelle, BYTE Columnist (1982:12 p242)

Desktop publishing didn't really make its debut until Apple announced the LaserWriter in January 1985 — John W. Seybold, Founder of The Seybold Report on Publishing Systems *(1987:5 p149)*