BASIC Answers to BASIC Questions

Since our BASIC first came on the market, we have had many questions regarding some of the more complex features of both our BASIC and Extended BASIC. The purpose of this document is to help clear up some of the confusion and to demonstrate some of the features which make BASIC and Extended BASIC so powerful.

This document can be divided into two sections: clarification of features found in both BASIC and Extended BASIC, and clarification of features found only in Extended BASIC. Each command, statement or feature references sections of the User's Manual where the initial discussion can be found. In order to be complete, several lines from the User's Manuals will be included in the discussions along with further explanation and examples. It is recommended that the user also rereads these sections referenced since not all of the information in those sections will be duplicated here. In this way the user may be able to understand the features in more depth.

It is our goal to try to present clear examples in both BASIC and Extended BASIC. The user is advised to follow the examples carefully and to try variations of these examples to further reinforce their proper use. The user is reminded that some of these examples may contain features not supported in their version of BASIC. That is, Extended BASIC examples should not be expected to work on a system supporting only BASIC! With this in mind let's start.

OUTPUT TO A PRINTER (Section 8.6)

(Section 8.6 in the User's Manual contains information referenced here.) Channel 0 has special meaning for the PRINT statement. Using channel 0 without using an OPEN statement when PRINTing is just like PRINTing to the terminal.

```
200 PRINT #0, "THIS IS A TEST"
```

will print THIS IS A TEST on the terminal, exactly as if the "#0," were not present.

Opening channel 0 allows you to send output to some other device such as a printer instead of the terminal. Using OPEN on channel 0 tells BASIC to read the file name specified as a file containing a printer driver routine such as PRINT.SYS in FLEX and use this new output routine whenever PRINT #0 is specified. For example,

```
10 OPEN "0.PRINT" AS 0
20 PRINT #0,"TABLE OF SQUARES AND SQUARE ROOTS"
30 FOR I=1 TO 100
40   PRINT #0,I,I*I,SQR(I)
50 NEXT I
60 CLOSE 0
```

Line 10 tells BASIC to use file 0.PRINT.SYS (SYS is the default extension when channel 0 is referenced) as the output routine on channel 0. Notice the OPEN OLD was not necessary. On line 20 we print a header line which is output through the printer routines found in the PRINT.SYS file. Line 40 will now print on the printer the values of the number (I), the square and the square root of the number (I). The CLOSE 0 statement will now make all following PRINT #0 statements act just as in the example at the beginning; all output will be displayed on the terminal until another OPEN AS 0 statement is executed. It is very important that the file specified in the OPEN AS 0 statement is actually a printer system file such as PRINT.SYS. If it is not, unpredictable results will occur, possibly crashing BASIC!

VIRTUAL ARRAYS AND RECORD I/O (Section 10.2-10.14)

There still exists some confusion about virtual arrays and record I/O. An example using both of these powerful features should aid in clarifying their use.

```
10 OPEN "EMPLOYEE" AS 1
20 FIELD #1, 20 AS NM$, 22 AS Z1$, 2 AS DI$, 64 AS Z2$
30 OPEN "CHANGES" AS 2
40 DIM #2, DV$(999)
50 FOR I=0 TO 999
60   GET #1
70   LSET DI$ = DV$(I)
80   PRINT NM$; " division changed to "; DI$
90   PUT #1
100 NEXT I
```

This program is designed to mass update the employee's division number of an employee file stored in record I/O form with a table of division changes stored as a virtual array. Line 10 OPENs the file EMPLOYEE on channel 1. Line 20 describes the file's format. Each employee record is a total of 108 bytes long; however, BASIC will use an entire sector (252 bytes) to store it. The two variables Z1$ and Z2$ are used to preserve spaces to position DI$ properly. Line 30 then OPENs the virtual array on channel 2. This virtual array is dimensioned as 1000 elements in line 40. The loop (lines 50 through 100) goes sequentially down the file and array, making the proper change and displaying a message to the terminal.

Realize that with record I/O we could have done the GETs and PUTs in a random order. The same program could have been written using a two dimensioned virtual array with the employee number in one field and the division change in the other field. Now the GETs and PUTs depend upon the employee's number. For example:

```
10 OPEN "EMPLOYEE" AS 1
20 FIELD #1, 20 AS NM$, 22 AS Z1$, 2 AS DI$, 64 AS Z2$
30 OPEN "CHANGES" AS 2
40 DIM #2, DV$(350,1)
50 FOR I=0 TO 350
60   GET #1, RECORD CVT$F(DV$(I,0))
70   LSET DI$ = DV$(I,1)
80   PRINT NM$; " division changed to "; DI$
90   PUT #1, RECORD CVT$F(DV$(I,0))
100 NEXT I
```

This example works just like the previous one except that the employee number (and record number) is found in the first column of the virtual array table while the division change is in the second column.

One more example just using record I/O with random access will help

to clarify record I/O. This example can be typed in and run because it will set up its own data. This example will use floating point numbers in a random file; therefore, there will be a difference between the standard BASIC and Extended BASIC version of this program. Standard BASIC uses only four bytes to store a floating point number, whereas, Extended BASIC will need eight bytes to store a floating point number. This means that in the example below, the

```
FIELD #3, 4 AS .  . . 4 AS  .  . .
```

should have the 4's replaced with 8's to run on the Extended BASIC versions.

```
10 OPEN NEW "TABLE" AS 3
20 FIELD #3, 4 AS SQ$, 4 AS SR$, 4 AS LG$, 4 AS EX$
30 REM set up arithmetic table.
40 INPUT "Size of the table"; SZ
50 FOR I=1 TO SZ
60   LSET SQ$=CVTF$(I*I)
70   LSET SR$=CVTF$(SQR(I))
80   LSET LG$=CVTF$(LOG(I))
90   LSET EX$=CVTF$(EXP(I))
100  PUT #3
110 NEXT I
120 REM now the table is set up.
130 PRINT "Enter a number between 1 and ";SZ;" , 0 to exit. ";
140 INPUT #0,NM
150 IF NM=0 THEN 200
160 IF NM>SZ THEN 130
169 GET #3, RECORD NM
170 PRINT NM;" squared is ";CVT$F(SQ$);" square root:";CVT$F(SR$);
" log:";CVT$F(LG$);" and exp:";CVT$F(EX$)
190 GOTO 130
200 CLOSE 3
210 END
```

This program will open the random file and set up the arithmetic values of numbers from one to a value selected by the user. Once the random file is set up, the user can randomly select number values from one to the size previously selected and display the square, square root, logarithm, and exp of that number. The user is invited to try this example and experiment with other variations.

THE USR FUNCTION (Section 11)

The USR function can be employed when the user needs to perform some function not found in the instructions of BASIC or Extended BASIC. For example, you have a system that includes salesmen numbers. For security you wish to encode these numbers and then, when they are needed, you may decode them. A very simple way to encode the numbers is to perform a shift or rotate on the bits making up the number. The next example shows how to call multiple USR functions to encode a number by rotating to the left and decode by rotating to the right. 6800 Example:

```
200 INPUT "NUMBER TO ENCODE OR DECODE",CD
210 UL=HEX("24"):UR=HEX("25")
220 XL=HEX("6E"):XR=HEX("00")
230 YL=HEX("6F"):YR=HEX("F0")
240 INPUT "ENCODE (E) OR DECODE (D)",TY$
250 IF TY$="E" THEN 300
260 IF TY$="D" THEN 400
270 GOTO 240
300 POKE UL,XL:POKE UR,XR:NU=USR(CD)
  . . .
400 POKE UL,YL:POKE UR,YR:NU=USR(CD)
  . . .
```

With an assembly program like:

```
       ORG     $6E00
ENCODE LDAA    $26    UPPER ORDER BYTE
       LDAB    $27    LOWER ORDER BYTE
       CLC            RESET CARRY TO 0
       TSTB           TEST THE SIGN BIT OF THE LOWER BYTE
       BPL     ROTAT1 IF BIT IS 0 THEN CARRY BIT IS ALREADY 0
       SEC            ELSE, THE CARRY BIT NEEDS SET TO 1
*****************
* THE CARRY BIT (THE  SAME AS THE SIGN BIT OF THE LOWER ORDER
* BYTE) IS SHIFTED INTO THE BOTTOM OF THE UPPER ORDER BYTE
*****************
ROTAT1 ROLA
*****************
* THE CARRY BIT (THE SAME AS THE SIGN BIT OF THE UPPER ORDER
* BYTE) IS SHIFTED INTO THE BOTTOM OF THE LOWER ORDER BYTE
*****************
       ROLB
       STAA    $26    THE NEW ENCODED UPPER ORDER BYTE
       STAB    $27    THE NEW ENCODED LOWER ORDER BYTE
       RTS

       ORG     $6FF0
DECODE LDAA    $26    UPPER ORDER BYTE
       LDAB    $27    LOWER ORDER BYTE
       CLC            RESET CARRY TO 0
```

```
       ANDB    #$01   MASK OUT ALL BUT LOWEST BIT
       BEQ     ROTAT2 IF LOWEST BIT IS 0 THEN CARRY IS 0
       SEC            ELSE,THE CARRY BIT=LOWEST BIT OF LOW BYTE
ROTAT2 LDAB    $27    RELOAD LOWER ORDER BYTE
****************
* THE CARRY BIT (SAME AS THE LOWEST BIT IN THE LOWER BYTE) IS
* SHIFTED INTO THE TOP OF THE UPPER ORDER BYTE
****************
       RORA
****************
* THE CARRY BIT (SAME AS THE LOWEST BIT IN THE UPPER BYTE) IS
* SHIFTED INTO THE TOP OF THE LOWER ORDER BYTE
****************
       RORB
       STAA    $26    THE NEW DECODED UPPER ORDER BYTE
       STAB    $27    THE NEW DECODED LOWER ORDER BYTE
       RTS
```

BASIC line 210 sets up the address where the USR function can find the memory address of the assembly routine; in 6800 this is at $24 and $25. Line 220 sets up the memory address of the first USR assembly routine, ENCODE, at $6E00. Line 230 sets up the memory address of the second USR assembly routine, DECODE, at $6FF0. Line 300 then POKEs the address where the USR function expects to find the memory address of the assembly routine. Line 400 also POKEs this address. In this way, the user can choose between one of several assembly routines to execute. The assembly code for both ENCODE and DECODE is documented; we will forgo any explanation at this time.

An example of the 6809 USR function would look about the same. Of course, the POKE memory locations and assembly language would be different. In 6809 Extended BASIC or BASIC the USR function expects to find the memory address of the assembly routine at MEMEND-2, and the USR function's argument can be found at MEMEND-4. If MEMEND is $7FFF, then the address of the assembly routine should be placed at $7FFD ($7FFF-2), and the argument is found at $7FFB ($7FFF-4). Only the BASIC statements will be included here; the 6809 assembly programmer can easily insert the proper code. 6809 Example:

```
200 INPUT "NUMBER TO ENCODE OR DECODE",CD
210 UL=HEX("7FFD"):UR=HEX("7FFE")
220 XL=HEX("C1"):XR=HEX("00")
230 YL=HEX("C2"):YR=HEX("00")
240 INPUT "ENCODE (E) OR DECODE (D)",TY$
250 IF TY$="E" THEN 300
260 IF TY$="D" THEN 400
270 GOTO 240
300 POKE UL,XL:POKE UR,XR:NU=USR(CD)
  . . .
400 POKE UL,YL:POKE UR,YR:NU=USR(CD)
  . . .
```

```
And some 6809 assembly code:
       ORG     $C100  START OF ENCODE
ENCODE LDA     $7FFB
       LDB     $7FFC
  . . .
       RTS
       ORG     $C200
DECODE LDA     $7FFB
       LDB     $7FFC
  . . .
       RTS
```

Furthermore, the programmer of USR functions should also adjust the MEMEND (check the user's manual for location in your version) value in FLEX. In this way, BASIC will not use the memory space of the USR functions to run the BASIC program. MEMEND should be set (through the monitor) to a value just under the USR functions' first memory location. For example, set it less than $6E00 in the 6800 example above; however, as MEMEND is set to $BFFF, initially, it does not need to be changed for the 6809 example above.

In order to load both the BASIC program and the assembled USR function, the user needs to use the FLEX "GET" command to bring the assembled USR function into memory and the normal BASIC "LOAD" to load the BASIC program.

Extended BASIC features:
INTEGERS vs. FLOATING POINT (Section 4.3)

In Extended BASIC the user may use either integer variables/constants or floating point variables/constants. A great deal of confusion has risen from the use of integers. Integer arithmetic will always work in whole numbers, no fractions are used as in floating point. Therefore, the results of integer arithmetic may be different than expected. For example, one may assume that 5/2 would yield 2.5; however, in integer arithmetic the .5 is truncated to leave only the whole part, 2. This means that

5 / 2 * 2 is 4.

The answer is 4 because 5/2 is 2 (not 2.5 in integer arithmetic!) and 2*2 is 4! 10/3 is 3 (not 3.333333...); however, 10./3. is 3.3333... This brings into the discussion the mixing of floating point and integer variables/constants. The computer calculates the result of arithmetic by grouping the operands and operators into groups of three based on the precedence of the operator, moving from left to right. That is, in A*B+C/D the A*B is calculated first, C/D second and the addition of the two sub-results last. The type (integer or floating point) of the two operands (or sub-results) determines the type of the result. If one of the operands is floating point then the result of the operation will be floating point. For example,

5. / 2 + 5 / 2 is 4.5.

Because 5./2 is 2.5, 5/2 is 2 and 2.5+2 is 4.5, the answer is 4.5. If the user is using values with only whole numbers, the integer type will be faster and requires less space (only 2 bytes). Once again the user is advised to experiment with the differences in floating point and integer arithmetic and the mixing of the two.

SCALE COMMAND (Section 5)

Another point of confusion lies with the SCALE command. Users have asked us both, why is it used and how is it used properly with SAVE and COMPILE programs.

Why is scaling used? A computer uses the binary (base 2) number system while humans use the decimal (base 10) number system. There are several values to the computer which cannot be accurately represented. For example, 1/3 (base 10) represented in decimal and 0.1 (base 10) and 0.01 (base 10) represented in binary are very common values; they are considered repeating digits and cannot be accurately represented in a finite amount of space, ie. 1/3 = 0.33333... in base 10 and 0.1 decimal = 0.000110011001... to the computer in binary. Because values such as 0.1 and 0.01 cannot be represented accurately in binary, this inaccuracy, as small as it may be, is carried or propagated through successive arithmetical operations. That is, by adding, subtracting, multiplying, etc. the value 0.1 (or any other values not accurately represented) a number of times, the small inaccuracy or error grows larger with each arithmetical operation. Therefore, to the computer, 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 does not quite equal 0.8! In situations where you may have several hundreds or thousands of computations, this error (called round-off error) could result in false results or in improper testing in IF statements. For example, if you were testing for equality of a computed result with a constant, due to round-off error, they may not be exactly equal! For example:

```
10 FOR I9=1 TO 1000
20   F=F+0.1
30 NEXT I9
40 IF F=100 THEN PRINT "EQUAL!"
50 END
```

Without setting the SCALE factor, 'EQUAL!' will never be printed; the test of equality will fail!

Now, where does scaling come in? If we could tell the computer to do something to 0.1 or 0.01 in order that it would accurately represent these values, we would no longer need to worry about round-off error. Scaling is the answer. With the SCALE factor set to non-zero, Extended BASIC scales all floating point values by 10**SCALE factor and rounds to a whole number. In effect all floating point numbers become integers; the fractional parts disappear. With a SCALE factor of 1, 0.1 becomes 1.0 internally and 0.29 becomes 3.0 internally (rounded to whole number 3). Using a SCALE factor of 1, the above program will now print 'EQUAL!'; the equality test will succeed. With a SCALE factor of 2, 0.1 becomes 10.0, 0.01 is 1.0, 0.29 is 29.0 and 0.184 is 18.0 (rounded). Please note, 0.01 is only 1.0 to the computer, internally used in arithmetic; the computer scales all values, performs arithmetic and then will print them out just as before scaling! A PRINT statement will still yield 0.01; however, 0.184 will be printed as 0.18 (with SCALE

factor of only 2) due to the rounding of the 4 in the thousandths place. Therefore, if you intend on using values like 0.0001 or 0.18574 in which you desire to preserve accuracy, then you need to set the SCALE factor accordingly. Note that the maximum SCALE factor is six (6).

Now that you know why a SCALE factor is used, let's look at the proper use with SAVE and COMPILE programs. As Extended BASIC converts all constants to their binary equivalent when the program is typed in or LOADed, it is not possible to change the SCALE factor while the program is in memory. Not only is it an error (#67) to change the SCALE factor while a program resides in memory, it would be worthless as the constant values have already been set. Only by first SAVEing the program to disk, changing the SCALE factor and then reLOADing the program (thus resetting the constants to binary values using the new SCALE factor) can a SCALE factor be changed for an Extended BASIC Source program (BAS extension).

Furthermore, as a BASIC Compiled program (BAC extension) cannot be LOADed. the SCALE factor at the time of COMPILE is stored on the disk with the COMPILED program. Subsequently, RUNning the program brings both the program and the SCALE factor into memory.

MORE PRINT USING EXAMPLES (Section 6.4)

PRINT USING is a very powerful formatting feature in Extended BASIC. We have been asked questions about its use and feel that a few more examples may help to clear up these questions.

STRINGS (BACK SLASH):

```
PRINT USING '\23456\', 'THE RAIN IN SPAIN FALLS'
THE RAI
Altogether seven (7) characters counting both of the back slashes!
```

NEGATIVE NUMBERS using the POUND SIGN:

```
PRINT USING "####", -235
-235
A minus sign can be printed before a number using the pound sign.
```

FLOATING DOLLAR SIGN $:

```
PRINT USING "$$###,###.##", 23.05
      $23.05

PRINT USING "$$###,###.##", 38293.4
  $38,293.40

PRINT USING "$$###.##-", -3.5
   $3.50
Note the leading spaces, right justification and floating dollar sign.
```

$ and * RESERVE ONE NUMERICAL SPACE:

```
PRINT USING '$$#.## WITH PROTECTED FIELD IS: $**#.##', 12.5, 12.5
$12.50 WITH PROTECTED FIELD IS: $*12.50
Note that both $ and * reserve a space for a numerical field.
In this case it is the tens position.
```

EXAMPLE CONTAINING MANY OPTIONS:

```
PRINT USING "THE BALANCE OF \23456789\ IS $$##,###.##- AND
$**#,###.##-", 'AJEX, INC.', -2345.7, 3568.91
THE BALANCE OF AJEX, INC. IS   $2,345.70- AND $**3,568.91
```

PRINTING FLOATING POINT NUMBERS (Section 6.4)

Many users have asked us how to print a large floating point number in a decimal format as opposed to the scientific notation. This can be done by using the PRINT USING statement and giving the floating point number the exact format you wish to use. For example:

```
PRINT USING "###,###,###,###.####", 1234567890.1234
  1,234,567,890.1234
PRINT 1234567890.1234
1.2345678901234E+09
```