MICROPOLIS USERS GROUP

MUG Newsletter # 2 - September 1980

Published monthly by the MUG, Buzz Rudow - Editor, 604 Springwood Cir., Huntsville AL. Subscription rate: $12/year, $2/single copy.

Gentlemen:

Another month - how time flys. I pity those poor columnists who have to put out a column every day. Until I get some feedback on what the membership wants to see, I'll just write about some of my investigations.

We won't have any huge problems transferring software among the membership. The variations we have are MOD I and MOD II disks, and Version 3 and 4 of the Micropolis systems. BASIC programs have the same syntax in both Versions 3 & 4, although the interpreters are different. Assembly language is potentially a bit of a problem since the entry points in MDOS changed, and people also tend to access routines in their lowest level operationg system. These problems are controlable. To go from MOD I to MOD II (or back) disks, all one has to do is have both drives and a MOD II controller board. We'll discuss more on the specifics if we decide to start a library.

The only other major concern is the input/output to devices other than the disks, specifically to the CRT. It was pointed out that my example last month of clearing the CRT was too simplistic. It would work for memory-mapped devices but not for external terminals. By using the function statement, we can still overcome the problem.

By the way, even if it's only good for memory-mapped video, I found a way for my BASIC programs to compute whether they are running on my SOL or Vector. RES location 16R0500 contains the line length of the screen. Again, if 'C' = the 'clear screen' command, and 'K' = the keyboard input buffer:

```
     IF PEEK(16R0500)=79 THEN C=4:K=1
     IF PEEK(16R0500)=63 THEN C=11:K=252
```

Thereafter, as I said, I use PRINT CHAR$(C) and IN(K). This only works, of course, if the screen size of your two systems happens to be different.

Those of you who are Systemation software owners already know that Systemation included our circular in their mailout for the latest CRUNCH program. That assistance should help our cause immensely. Systemation's Bob Zayle has already been immensely helpful. While I realize that aiding in a project designed to increase activity with Micropolis is self-serving to a Micropolis software house such as Systemation, Bob has not just been cooperative, he has spent literly hours on the phone with me. Anyone who's in a small business knows that that sort of diversionary time is not easy to come by and still stay in business. Bob, the MUG appreciates your help and thanks you profusely.

One of Bob's hints pertains to the GOTO statement. For illustration, I supposed I had a program to separate disk records from one file into three new files. Separation is dependent on the contents of some variable in the original file. The program (TEST1) doesn't do any disk reads or writes. The variable J is arbitrarily incremented to simulate the decision variable in an incoming record. The incrementing of the variables A, B, and C simulate the writing of output files. I do this sort of thing, and indeed use the ON - GOTO statement.

The purpose of all this is that the GOTO supposidly takes a great amount of time to execute. Micropolis BASIC goes through the program, front to back, each time to find the line referenced by the GOTO. The test was to run a short program by itself and then again when it was appended to the physical end of a large program. My large program was 16700 words long. My execution time, shown in TABLE 1, indeed amazed me - 33.7 seconds alone, 281.2 seconds when on the end - a 734% increase in execution time. So I tried TEST2 with no GOTO's. Indeed, though the stand-alone version ran some 10% slower than the stand-alone TEST1, the appended version ran almost as fast as the stand-alone. (Why did it run slower? - I don't know)

As an afterthought I tried TEST3, which is another equally acceptable "proper" way to program the situation. Theoretically, you shouldn't waste time doing the extra IF's if you already have the answer. As expected, TEST3's stand-alone execution time was equal to TEST1's. What was unexpected was the appended execution time. TEST3 was slightly faster than TEST2 even though it had GOTO's. The ON - GOTO was the real culprit. I wasn't even sure the GOTO's were causing any trouble. TEST4 seems to answer that. Merging the test into the large program indeed caused a 42% increase in execution time over the stand-alone version.

The conclusions to be made from all this seem to be:

1. Always stay away from ON - GOTO's.
2. If possible, put your code in-line rather than doing a GOTO.
3. A routine in a short program runs faster than the same routine in a large program.
4. As a general rule, keep your "most-executed" routines, especially those with GOTO's, at the physical front (low line numbers) of your program.
5. Put all your comments and non-time-critical routines (menus, etc.) at the physical back of your program.

```
20000 ! TEST 1                          20000 ! TEST 3
20005 J = 0                             20005 J = 0
20010 FOR I = 1 TO 1000                 20010 FOR I = 1 TO 1000
20015 J = J + 1: IF J > 3 J = 1         20015 J = J + 1: IF J > 3 J = 1
20020 ON J GOTO 20025, 20030, 20035
20025 A = A + 1: GOTO 20040             20025 IF J = 1 A = A + 1: GOTO 20040
20030 B = B + 1: GOTO 20040             20030 IF J = 2 B = B + 1: GOTO 20040
20035 C = C + 1: GOTO 20040             20035 IF J = 3 C = C + 1: GOTO 20040
20040 NEXT I                            20040 NEXT I
20045 PRINT I, A, B, C                  20045 PRINT I, A, B, C
20050 END                               20050 END
```

```
20000 ! TEST 2
20005 J = 0
20010 FOR I = 1 TO 1000
20015 J = J + 1: IF J > 3 J = 1
20025 IF J = 1 A = A + 1
20030 IF J = 2 B = B  + 1
20035 IF J = 3 C = C + 1
20040 NEXT I
20045 PRINT I, A, B, C
20050 END
```

```
00001 GOTO 20000
00010 GOTO 1000
01000 GOTO 20015
05000 GOTO 10
20000 ! TEST 4
20005 FOR I = 1 TO 1000
20010 GOTO 5000
20015 NEXT I
20020 END
```

TABLE 1 - TEST EXECUTION TIMES (SECONDS)

Execution Times:	TEST 1	TEST 2	TEST 3	TEST 4
Alone	33.7	36.8	33.7	13.7
Merged	281.2	37.3	37.2	19.5

Let's take a look at the SIZES statement. It is formated

```
    SIZES (r,i,s[,n])
```

where r = 3 to 30
i = 2 to 29
s = 1 to 250, and
n = a number designating the arbitrary end of the program.

In the cases of 'r', 'i', and 's', the integer signifies the number of words the computer allocates to the 'real', 'interger', and 'string' variables. Manipulation of 'r' and 'i' can cause significant impacts to your programs. To a lesser degree, so can 's'. The default values are r=5, i=3, and s=40.

Starting at the bottom, if 'r' is set to 3, the computer allows you a real number of 4 significant places - and it truncates. That is, an input of '.123499999' gives '.1234'. The extremes look like this:

```
    r = 3; + and - 9.999E60
      = 4; + and - 9.99999E60
      = 5; + and - 9.9999999E60
      = 6; + and - 9.999999999E60
```

The maximum size of the number stays more or less constant, but the precision, or accuracy, doesn't. The bigger the size of 'r', the more precise your answer. The system doesn't tell you if you are truncating. It does tell you if you exceed the extremes, but that is a very large number. To illustrate the impact, suppose you want to add the numbers 10,321.49 and 538.70. If 'r'=3, your answer will be 10,850.00 - not exactly what you expected. To get the accuracy you desire, you have to set 'r'=5. That will get you 8 significant digits, enough for 999,999.99. Setting 'r' to 4 would have truncated the hundreths place.

If 'i' is set to 2, the computer gives you an integer number of four digits, although the most significat digit can't exceed a value of 4 or 5. The extremes are:

```
    i = 2; -5,000 to +4,999
      = 3; -500,000 to +499,999
      = 4; -50,000,000 to +49,999,999
      = 5; -5,000,000,000 to +4,999,999,999
```

As you can see, the size of the number is increasing by 2 digits for each single step increase in the SIZE value. That calculates to a maximum size of 58 digits, which should be sufficient for most of us. Exceeding the allowable size in an input statement will cause the system to give you an error. That is not the case if you are doing math within your program. If 'i'=2, then 4999 + 1 = -5000 - Yes, minus 5000. Be sure you SIZE your integers large enough to take the biggest number you will generate.

There are other considerations besides size and exact precision. Memory size is one. One might conclude from the above discussion that you should just open up the SIZES to (30,29,250) and forget about it. However, in addition to specifying the number of digits and the precision, the SIZES numbers are stipulating the number of words used to represent each occurance of any variable of that type. In the default mode, SIZES (5,3,40): DIM A(2000) would allocate 10000 words (5 times 2000) for the A array. I use this size of an array when I'm sorting mailing files since I can get greater than 2000 logical records on a Micropolis disk. I obviously don't want to arbitrarily use SIZES (30,29,250) or my array would occupy 60,000 words (30 times 2000), a bit much for a 48K machine. Actually, the 10,000 word allocation is a bit much since the interpreter takes a minimum of 22,272 words, and the rest of the program takes space. If I set SIZES to 3 (for real numbers) I would truncate the last digit of the 5-place ZIP code. But by setting 'r'=4, I keep the required accuracy (up to 6 places) and cut the allocation to 8,000 words. I could also use an integer array and set 'i'=3 and save another 2000 words.

There is an additional problem associated with real numbers. I have one billing program that adds a 1.5% monthly interest charge to any upaid balance. I use a format statement for printing the bill that just prints the "normal" digits. That is, for a balance of 37.22:

```
        UNPAID BALANCE       $ 37.22
        INTEREST CHARGE          .55
                             ------
        BALANCE DUE          $ 37.77
```

But the computer knows that the balance really is $37.7783 (37.22 times .015 = .5583). When the customer pays the bill the computer still has a value of .0083 in that variable. My problem was that I scaned this variable for "greater than zero" amounts as in indication of unpaid balance. Even though the bill was paid, the program would cause a new bill to be generated with a $0.00 UNPAID BALANCE, and a .00 INTEREST CHARGE. You don't solve the problem by just throwing away the "surplus" billing statements. Left alone, the value eventually generates printable errors in the cents digit.

There are several ways of getting around the problem. You can work strickly in integers. I opted to leave variable as real but always make adjustments when multiplying. If A=UNPAID BALANCE and B=INTEREST CHARGE:

```
    B = A * .015:!            .5583 = 37.22 * .015
    B = B * 100:!             55.83 = .5583 * 100
    B = INT(B):!                55. = INT(55.83)
    B = B/100:!                 .55 = 55/100
```

or, putting it together: B=(INT((A*.015)*100))/100

I really haven't gotten into the topics I mentioned for this month. Systemation hasn't released their BASIC compiler yet. I did receive their SORT/A, and while it's every bit as good as predicted, I haven't documented my experience yet. It sure is fast, though. I did speak to Micropolis about their hard disk. While it's "kind-of" released, it will still be a couple months before you'll see ads for it. They also are releasing a double-sided disk. To meet my deadline, development of these subjects must wait another month. Oh, yes - the Micropolis Newsletter, after a year and a half of hibernation, is soon to be released. We're promised that it will be sufficiently full of news to make up for the delay.

9/1/80