Some Hints & Tips for writing efficient code.

Although there is merit in the argument that if you need speed, don't use an interpreted language, any code you write that is intended to be reasonably long-lived (i.e. you actually save it with the intention of using it again some day) should be honed to be efficient.

Much of what follows is idiosyncratic, fairly heavily pedantic and not terribly important but it scratches my efficiency itch. I come from a Z80/68000 background and older CPUs were starved of memory & power and it taught me to be frugal. Not much of the following really applies to compiled code and modern CPUs but there are elements of “sanitary” code development that will make you feel warm and fuzzy all over. It is worth mentioning that modern code suffers from a phenomenon colloquially known as “bloat” - this is largely the effect of developers not writing compactly (or even sloppily) because they don't have to care in these days of immensely fast multi-core CPUs and gigabytes of memory. Am I being too hard on them? Perhaps. In the days of slower systems, you could write something and see that it just wasn't good enough and so you'd have several stabs at it each time getting better (hopefully). Modern machines are so fast that even garbage runs in a twinkle so maybe developers don't get that kick in the pants when something they have written clearly runs like a three-legged donkey - the impetus to write excellent code just isn't there now. Personally, I get a buzz out of making code as good as I think I can get it, maybe revisiting code months later just for the mental exercise… Old habits and all that.

There are plenty of methods - some open to discussion, that can help. These range from the simple to downright cryptic, and it's not a new concept. Simple tweaks in the way you do things can shave quite a bit off the cycle times for your code - and that will be the over-arcing theme of these hints; to get code as quick and compact as possible, bearing in mind also that small isn't always fast… we'll come to that.

This article is aimed at the MicroMite range of micro-controllers but the ideas are fully migrate-able across all the MMBasic platforms and beyond. On the smaller micromites, some tasks can be a big ask for an interpreter at 48MHz (or slower). Some of the following tips got an RC4 encryption algorithm usable - it was struggling to do

```
receive >> un-encrypt >> process >> encrypt >> transmit
```

in under 1.5 seconds. It's still tight for long strings but it's usable now. So these tips can help and even have proved vital in some instances - do not dismiss as mere pedantry too lightly.

This isn't an exhaustive list; more a collection of hints and suggestions.

"Heal thyself, physician"

Many examples of my own code published in this library might not adhere too closely to these guidelines. This is likely for two main reasons:

1. These habits have evolved since I started publishing and are part of my own development.
2. I have not honed the code too much so as to leave it easily understandable.

Often when I use the code in anger, it will bear just a passing resemblance to any in a wiki article here.

Discuss, critique, contribute.

Keep your variable names relevant and punchy

Self documenting code is greatly helped by having variable names that describe what they are used for and it's a great idea... but... it can be over used, no-one will mind if you call a general purpose counter “x”. Consider the following:

```
For MyBigLoopCountVariable = 1 To 10
Print MyBigLoopCountVariable
Next
```

doesn't improve clarity over

```
For x=1 To 10
Print x
Next
```

Not to mention the smaller code footprint.

Short variable names are fine where applicable (and a tiny bit faster to parse too, so will shave off a few mS in tight loop with thousands of iterations), but the following is crying out for good variable names.

```
a(m,j)=c/d+1
```

looks a lot better as

```
ColorMap(current,pointer)=OldColVal/daynum+1
```

you stand a good chance of understanding what that line is doing just by looking at it.

Some insight into variable names

In tight/repetitive loops where timing is important, it helps to understand how MMBasic defines and accesses variables - it might help shave a few microseconds off each time through. Remember, the following refers to MMBasic as used in MicroMites not those later versions that use hash tables. MMBasic uses a variable descriptor table. Each variable has a 64 byte entry in VARTABLE with pointers to chunks of the heap for arrays and strings (integers and floats are stored in the descriptor). There is no ordering or reserved variable space in the table - each variable has to be searched for by name, every time you use it (BBC Basic had 26 reserved spaces for the 26 integer variables

```
a-z
```

which meant it didn't have to search for them it “instinctively knew” where each was - this was a

clever accelerant for the interpreter). With MMBasic, you can leverage the internal mechanism a tad by defining commonly used variables before obscure/rarely accessed ones, the most commonly used ones first, in the order they are used if you can. E.g. If you have a variable

```
n
```

that is used in counting loops everywhere in your code, define it early on so the interpreter doesn't have a hard time finding it when it needs to. Conversely, a variable you use twice in some bit of code that executes once every 10 minutes, define it last maybe so MMBasic doesn't have to skip over it in its search for

```
n
```

all the time. MMBasic parses variable names 4 characters at a time so there is no reason to arbitrarily reduce variable names unless you want to (to save code space?), e.g. to use

```
t
```

instead of

```
trev
```

, but it's a big reason not to use

```
trevor
```

(it needs two rounds of parsing) unless the longer name brings something to your code.

Don't calculate things twice - use a variable

If you make the processor do some work that you are going to need later, save the result and don't repeat the effort. The following forces some string slicing and evaluation to be done twice.

```
If Val(Mid$(a$,2,2))=17 Then
Print "The value is ";Val(Mid$(a$,2,2)); " in that position
EndIf
```

Variables are much faster than evals (ish... see below) Better to have

```
x=Val(Mid$(a$,2,2))
If x=17 Then
Print "The value is ";x; " in that position
EndIf
```

Wait! Don't use a variable

Consider this in a Sub or Function:

```
Local Integer x
x=t<<b
```

… and then using x in the places where you had the calculation, is definitely slower than simply using

```
t<<b
```

if only a couple of times because the overhead of variable management doesn't really play-off well against the speed of simple integer maths done once or twice.

The point I am trying to get across is; culture the mindset to play with the options and see which gives most benefit, don't necessarily accept the first thing that you think of for anything more than 'quick scribble' code.

On Absolutes and CONSTs

While on the subject of variables, don't forget CONSTants either - handled just like variables but obviously you can't change the contents. It takes a lot less effort to look up a value from a CONST than for the parser to work it out from your code, so try not to use absolute values if you can avoid it. Another tangent! when specifying absolute values, consider the parser… Us humans work really nicely in base 10, but processors don't. Any absolute value has to be converted into CPU binary each time it is encountered. Take a look at this silly code:

```
dim integer n

timer=0
do
 n=n+1
 if n=100000 then exit do
loop
? timer
```

Now consider where n is incremented. 1 is the same in all bases, but you can shave a lump off the increment by specifying a computer friendly base (even given the extra two bytes being interpreted). Here are some timings for various bases in just that one tiny line: 18268mS with n=n+1 17884mS with n=n+&h1 17841mS with n=n+&b1 17834mS with n=n+&o1 all the above mean the same thing but you can shave almost a second off that loop with a radix. Why octal is marginally the fastest I can't say, but I suspect it is down to the really easy “shift and add” method to build the ASCII representation into true binary.

This isn't the end though; that comparison line duffs-up the parser as well. Instead of using an absolute (even with a radix), how about using a CONST for the comparison? They are already stored in binary and very much faster to use. Going back to our

```
n=n+1
```

worst case above, by using a CONST, we shave almost 5 seconds off the same loop! Just 13839mS with

```
CONST c=100000
```

and then later

```
if n=c
```

roughly 25% faster. You could go even further and use another CONST for the increment value - I'll leave you to experiment with that.

In summary: Try not to use absolutes - use CONSTs unless you have a good reason and if you want/have to use absolutes, consider specifying them in different radices to shave off valuable mS. The key is balance: All of the above is largely meaningless outside of bits of code that run (or get called) over and over. And there isn't much point using a radix for the CONST (unless it adds something). It is only executed once in your code and I doubt you'll notice a few uS saved at boot time.

Multiple tests in IF statements

The If statement could be the arguably most used construct. It makes a decision and allows code to “look” intelligent by reacting to different situations (the much vaunted AI is largely just a mass of machine generated IF statements). The statement evaluates an argument and boils it down to a 1 or 0… true or false. Often time, the AND and OR operators are used to make the tests deeper but they can complicate a simple test and slow down the decision making. Try to make the arguments “aware” of what they are working towards. Consider the example we have to test the condition of a switch but only if it is daytime. We might write something like:

```
If DayTime(Now())=1 And SwitchPressed=1 Then...
```

but remember the whole argument has to be evaluated. The switch is tested as well as seeing if it is daytime. In reality, if it isn't daytime, we don't give a hoot about the switch so you can help the parser by splitting things down. Consider:

```
If DayTime(Now()) Then
    If SwitchPressed Then
```

Yes it takes more space in your code, but it will be a little faster. If this is executed hundreds of times a minute, all those savings add up. This method allows the parser to immediately skip the whole testing of the switch state if it is night time. Note also we don't need to test for

```
=1
```

. The parser is a trusting soul and it believes everything is true unless you say otherwise, so “

```
If DayTime() Then
```

” is functionally identical to “

```
If DayTime()<>0 Then
```

” but a little bit faster because no comparison (to 1) is made. Worth mentioning that every number except zero is true so you can use this trick for any non-zero condition.

While we are at it, do the quickest, simplest, most-unlikely things first. Which takes more time to execute? Which is the least likely thing to happen that can quickly disqualify the rest of the test? etc…

As an aside, this last aspect is the subject of an entire IT effort when streamlining databases; Consider a hypothetical world population database with the names and nationalities of every human in it (massive assumptions made to keep the numbers simple). Suppose now, we want a list of every Chinese male. Which of the two “queries” below is the better? By better I mean, fastest and lowest “cost” to the database engine.

```
 Select NAME From POPULATION Where SEX='M' And NATIONALITY='CH';
```

or

```
 Select NAME From POPULATION Where NATIONALITY='CH' And SEX='M';
```

Both results are the same and on the face of it, so are the two queries, but the way most databases work is through reducing sets. It starts out with a big block of data, then strips away the stuff it's not interested in to result in another block… and so on, until you end up with the subset you are interested in. The first searches our DB of 7 Billion records for all the males, (we'll assume it's 50%) and then has to search 3.5 Billion records to determine the Chinese members. 10.5 Billion records examined. The second example searches 7 Billion records, but after it only has to search a further 1 Billion records - 8 Billion records total. The first part of the search is unavoidable, but the second dispensed with 70% of the work - just by thinking about the task at hand and shaping the query accordingly, which represents a saving of almost 25% over the entire workload. This is a very blunt example but I hope it illustrates my point - get rid of the most obvious stuff first and deal with the intricacies after.

So armed with the logic of our database example, going back to our daytime switch-test earlier… Each time through that test during the day, the

```
If Daytime()
```

will be true, whereas we might reasonably assume the switch is not-pressed a lot less than all of the time - we can use that bit of knowledge to quickly disqualify the rest of the test, resulting in further reducing the load by reversing the two IF statements, thus:

```
If SwitchPressed Then
    If DayTime(Now()) Then
```

The functionality of the code is unchanged but we don't waste, 50% of the day on a meaningless test each time through that code. If it is running on the main thread (i.e. as part of a state machine - it does look very “statey”) then we have immediately reduced the time the processor spends before it

decides there's nothing to do and skips over any conditional activity - that DayTime() function and its argument could be very CPU hungry, so best to use it only when we have to.

Testing for permitted characters in a string

So you are asking the user to make a choice… Yes, No or Cancel… you might prompt them to enter only Y, N or C. The following code is a bit contrived but proves the point. By the way, don't use INPUT for this type of thing. It's two key-presses instead of one and is “blocking” - your program loses control while waiting inside the Input statement. INPUT is fine for scribble code but it is telling when used in a finished item.

```
Do
a$=Inkey$
Loop Until a$="Y" Or a$="y" Or a$="N" Or a$="n" Or a$="C" Or a$="c"
```

Twelve uses of the evaluator! OK, that is very contrived and we all know how to use UCase$ (right?) so you might write it as:

```
Do
a$=UCase$(Inkey$)
Loop Until a$="Y" Or a$="N" Or a$="C"
```

Six evaluations; halved the load. Happy with that? Remember what we said above about all the factors in an argument having to be evaluated. True if you are waiting for the user, the processor is likely twiddling its thumbs so the delay in processing is unimportant. But we are looking for efficiency. How about:

```
Do
a$=UCase$(Inkey$)
Loop Until Instr(" YNC",a$)>1
```

This is ultra efficient, tiny code and only a single evaluation - you could take it even further and make your test string a CONST/variable. It is also dead easy to add other options… you just put them in the test string in the Instr() function. It works by testing any returned character against a list of allowed ones. As far as Instr() is concerned, the empty string, “” (i.e. no key pressed) is always position one (that can throw people). The first character in the test string (in this instance, the space) is also in position one, so practically you can't tell if the user pressed that character or nothing at all, which is why we “burn” that position with a dummy… the really useful stuff starts at position two. Thus, if we return a value greater than 1, we have a valid keypress. Using Instr() to return the number of the character is very fast because we are using a building block (so-called primitives) of the language to do the work rather than doing it ourselves - always try to make the primitives do the work for you if you want fast code.

**A quick word about Inkey$** It gets the first key pressed from the buffer. It is important to remember that key is gone once Inkey$ has taken it, by which I mean, unless you take steps to save it… Notice above we assigned the value to a variable? Why wouldn't this be OK?

```
If Ucase$(Inkey$)="Y" or Ucase$(Inkey$)="N"
```

Don't put comments and white-space in the execution path unless you have to

I am a great believer in commenting and indenting your code to make it easy to follow. In a compiled/assembled language, it really doesn't matter where that happens, but in an interpreted language like MMBasic, all that stuff has to be skipped over to get to the code - this takes time (consequently, in the divisive argument of Spaces vs TABs; I am firmly in the TABs camp as it reduces code size and the number of white bytes to skip). Consider the following:

```
For n=1 To 1000
'does something
a=a+1
Next
```

looks innocuous enough, but each time through that loop, the comment must be skipped over and that takes time. This is better but functionally identical

```
'does something
For n=1 To 1000
a=a+1
Next
```

On the subject of comments, keep then succinct. Only where necessary - they take up space and slow code wherever they are. try to avoid white-space before or after a ' unless it helps in some way.

Don't make Subs or Functions for things you do only once if you can avoid it

Each time you branch to a Sub program or Function, it takes some time to set up the working environment, run your code, tidy up and come back.

If you use a piece of code only once, put it in the main stream of the code unless it really helps, i.e. it is only executed once in a while or the code depreciates the readability of the main stream - it is easy to isolate it and make it into a Sub later if you need to e.g. you end up calling it from somewhere else as well or it just starts to “fog-up” the program flow.

Consider if you are writing a sort algorithm that needs to swap two variables. This will likely be called thousands, if not tens of thousands of times on a reasonably sized list. The swap is only performed in one place. Yes it looks nice to have a Swap function in your code (MMBasic has no native Swap but there is a CSub which is really quick), if each time you call it, you waste just 500uS with all the house-keeping, over 2 thousand swaps that whole sort routine is now running a second slower than it might. Do you think that 2000 is an unrealistic number just to prove a point? In a bubble sort, a list of 100 items in worst case order will take 5000+ swaps to sort - just 100 items! At 500uS per swap, you have just added $2^1/_2$ seconds to the sort time simply by branching out to a nice Swap function - and that doesn't include *any* computation towards the task.

A moment of hypocrisy

When I publish my work (at least only snippets, functions etc), I really try not to use cryptic code but there are times when a simple Peek or Poke can really make a big difference. Beware of them though; beyond being very cryptic, they can bring incompatibilities between platforms.

The following are two very specific alternatives that will shave a good portion off specific string slicing actions.

When you are moving progressively through a string examining the ASCII codes of single characters. Consider this:

```
for n=1 to len(a$)
x=Asc(Mid$(a$,n,1))
Next
```

fairly obvious what we are doing and pretty much the standard method of obtaining the character code for each character in the string. Understanding that strings are simply long lists of these ASCII codes in the first place helps to derive a faster way of dealing with this:

```
for n=1 to len(a$)
x=Peek(VAR a$,n)
Next
```

I know I said I wanted to avoid cryptic, but this simple Peek directly returns the `number` of the pointed character - precisely what you want and exactly the same functionality as the first example, but only a single function and hugely reduced complexity for the evaluator. In fact, it's about 16% faster; With a 255 character string, a 48MHz MicroMite take 36mS and 30mS respectively. Dropping the CPU to 5MHz exaggerates the difference - 376mS and 300mS. A very healthy acceleration, but wait there's more…

When shortening a variable from the left, it would be very reasonable to have (assuming you want the leftmost 128 characters):

```
a$=Left$(a$,128)
```

But knowing the structure of strings (and that offset 0 is the length byte) you could easily have

```
Poke VAR a$,0,128
```

This simple tweak is an amazing 33% faster because it eliminates the string evaluator entirely. In a loop of 1000 iterations on a 48MHz MicroMite, the timings were 146mS and 98mS respectively.

The above two examples are reasonably hardcore and very specific to MMBasic, but they work across all current flavours of MMBasic and pay immediate dividends. You are also getting used to some of the more “low-level” functionality. Legibility can be hard to maintain in such circumstances and I would encourage you to avoid it where speed is not an issue, but if it gets you thinking then my work is done.

There are more “super-tweaks” in this vein, hopefully these two inspire you to explore a bit. Play around with the low-level stuff and have fun learning the inner workings and getting slick code in the process - even though it may not be terribly readable. **Save your program before running it because this stuff can cause processor exceptions**.

Stare at your code Re-read it, polish it, streamline it. Be proud that it is as efficient as you can get it but don't be resentful when someone shows you a better way. That is free learning! Remember: efficiency is what you determine it to be, not just small code, not just fast - it has to be a balance of the two that you like and can give an approving nod to.

Consider a Cosine calculation. You could write a tiny routine using Chebyshev polynomials to calculate the Cos() of any angle. It will be small - but it will be slooooow. If you analyse the requirement, you might find that you only need to calculate to a tenth of a degree… Now you could make up a table of 3600 (10 steps for each of 360 degrees) with all the values for the relevant Cosine and simply look the result up by jumping to the right point in the table: the input degrees, multiplied by 40 (10 for each degree * 4 for each floating point number) added to the table start address and you have your answer. No real calculation at all. Super fast… but also super big; 14.4K of memory taken up. You could reduce this by 75% by performing some massaging of the input angle based on its quadrant but now you are complicating things so the efficiency of speed might no longer play-off against the inefficiency of memory.

This, incidentally, is the eternal conundrum of computing; you can have it big and fast or small and slow.

Probably more to come…