AFTER, EVERY, AT, REMAIN Flexible State Machine Timers using Semaphores.

A common requirement is for sections of code to be executed to a schedule or after a period of time. The following uses an array of flags and counters to keep track of (in this case but easily tweak-able) 10 timers, with option for them to be single-shot or repeating. The timers can be stopped or temporarily disabled. To check if a period has elapsed, simply check the flag and perform your desired action if it is set (you need to reset it as part of this). This is a key requirement for State Machine Methodology.

The Subs/Functions are inspired by the Locomotive Basic commands of the same names and make executing sections of code asynchronously a breeze. MMBasic provides four timers but they expect a sub program to be called each time and they have to be handled (i.e. if they are one-shot) by the called sub. The following code uses a single MMBasic timer but then expands this to practically unlimited timers and really easy to set one running. The main loop simply checks for the relevant flag and takes action (calls a Sub, changes the state of an output pin etc...) if it is set.

After Set a semaphore after a period of time. E.g. After 30 seconds **Every** Set a semaphore at the same period repeatedly. E.g. Every 10 seconds **At** Set a semaphore at a specific DateTime. E.g. At 09/09/2019 05:25:30 **Remain** Return the remaining time (i.e. before it was due to set the semaphore) and optionally stop the timer. **DI & EI** Temporarily Disable or Enable all timers.

Pros:

- Easy to configure a one-shot timer.
- Easy to re-use timers for different sections of code.
- Retrieve the remaining time on a timer and optionally stop it.
- No need to worry about handling SETTICK - No dedicated Sub to handle an event - just do it inline when it is convenient.
- Gives much greater scope for the number of timers.
- Multiple ticks from the same counter give only a single event - No event “storms” for slow-running code.
- Manageable program interruption - It is totally down to the program flow - nothing kicking off “behind your back”.
- No implicit priority to timers, it is down to how the flags get checked.

Cons:

- Timer resolution is not as good as SETTICK on slow systems - very small delays are likely to be inaccurate, e.g. waiting one second when the service period is 1000mS.
- As the internal timer is used to determine the time periods, very long periods may be inaccurate due to clock drift (see clocktrim in the MicroMite manual). Note this is no worse than the SETTICK command. This may become particularly noticeable when using At() - A little work could provide a better solution if it becomes critical.

The following provides a core timer handler and three Subs to set repetitive or one-shot timers and a method to stop a timer (and return the remaining count before it was stopped).

Dependencies:

- Bit Manipulation Functions Bits 0 - 9 in the Flags field correspond to timers 0-9... tweak as necessary. Bit 10 is the global timer disable bit (=1). In the disabled state, timers are not decremented and any time spent disabled will be effectively added to running timers. E.g. if a timer is due to expire after 10 seconds but spends five seconds disabled, the total delay will be 15 seconds. DI/EI should be used to stop the timers when some critical process is taking place to prevent state changes.
- Three global variables are required, two integer arrays and a simple integer to hold the flags.
- UnixTime required if you intend to use the At functionality

Variable Descriptions:

TMRctr() maintains the count for a given timer TMRini()holds the timer type and its initialization value.

```
    - bits 0-61 store the initial value
    - bits 63&62 store the format of the counter:
        0=disabled
        1=(actually 0x4000000000000000) indicates a one shot "AFTER x" type
counter
        2=(actually 0x8000000000000000) indicates a repetitive "EVERY x"
type counter
```

As written below, it provides 10 timers which should be loads for just about any use, but this can be tweaked as you see fit.

The CoreTMR needs to be set running with the SETTICK command - timer counter values are a multiple of the interval used here. Try not to be too aggressive with small/fast intervals but the larger the interval the greater the loss of resolution. Find something that works for you but remember to adjust your timer values accordingly e.g. if you want ticks 10 times a second, the setitick must be set to 100 - and the values used in AFTER & EVERY would need to reflect the increase in resolution -in the above example 1 second becomes 10 counts etc.

Example:

```
Preamble:
    Option Base 0

    Dim Integer TMct(9),TMin(9),Flag


    SETTICK 1000,CoreTMR,4' service timers every second
```

```
Main:
' example usage for the main loop
' gives the user 10 seconds to hit a key (using timer 1)
    Print "Quick Press a key!"

    After 10,1 ' 10 seconds using timer 1
    Every 1 ,2 ' count 1 second intervals using timer 2
    Every 2 ,3 ' count 2 second intervals using timer 3



    Do
        a$=Inkey$
        If FlagTest(2) Then FlagRes 2:Print "x"' output an x while we wait
        If FlagTest(3) Then FlagRes 3:Print "y"' output a y  while we wait

    Loop Until FlagTest(1) or a$<>""

    x=Remain(1)' stop the timer and get the remaining time

    If FlagTest(1) Then ' the timer expired?
        Print "Too late!"

    Else ' otherwise someone pressed a key
        Print "that was close. There was only ";x;" seconds left!"

    EndIf

    DI
    For n=1 To 3: x=Remain(n): Next
    EI

    End




' The subs
    Sub CoreTMR ' needs to run on a ticker at whatever interval is required
- this is also the multiple of timer counts
        If FlagTest(10) Then Exit Sub
        Local Integer n
        For n=0 To 9
            Select Case TMin(n)>>62 ' extract the top 2 bits
                Case 0,3 'disabled or invalid
                    FlagRes(n)
                Case 1' AFTER
                    If TMct(n) then
                        TMct(n)=TMct(n)-1
                        If TMct(n)=0 Then FlagSet(n) ' indicate the timer
has expired
                    EndIf
```

```
                Case 2' EVERY
                    If TMct(n) then
                        TMct(n)=TMct(n)-1
                        If TMct(n)=0 Then
                            FlagSet(n) ' indicate the timer has expired
                            TMct(n)=TMin(n) And &h3FFFFFFFFFFFFFFF ' reset
the timer
                        EndIf
                    EndIf
            End Select
        Next
    End Sub


    Sub After(Interval As Integer,Tmr As Integer)' starts a one-shot timer
        TMin(Tmr)=Interval OR &h4000000000000000:TMct(Tmr)=Interval And
&h3FFFFFFFFFFFFFFF:FlagRes(Tmr)
    End Sub

    Sub Every(Interval As Integer,Tmr As Integer)' starts a repetitive timer
        TMin(Tmr)=Interval OR &h8000000000000000:TMct(Tmr)=Interval And
&h3FFFFFFFFFFFFFFF:FlagRes(Tmr)
    End Sub

    Sub At(dt$,Tmr As Integer)' trigger at a specific date/time - assumes
the core timer is running in seconds
        'dt$ must be a datetime formatted as  dd/mm/yyyy hh:mm:ss
        Local Integer x
        x=UnixTime(Now())-UnixTime(dt$)
        If x<2 then
            FlagSet Tmr' if he difference is less than 2 seconds, trigger
the alarm immediately
        Else
            After x,Tmr' otherwise set a one-shot for the difference
        EndIf
    End Sub

    Function Remain(Tmr As Integer, opt As Integer) As Integer' returns the
remaining time in the counter.
        'if opt=0 (or omitted) timer is stopped
        'if opt<>1 timer remains running
        Remain=TMct(Tmr)
        If opt=0 Then TMin(Tmr)=TMin(Tmr) And &h3FFFFFFFFFFFFFFF
    End Function

    Sub DI
        FlagSet 10
    End Sub

    Sub EI
```

```
        FlagRes 10
    End Sub
```

All the functions are inspired by some innovative (for the time) namesakes in Locomotive basic as used on the Amstrad CPC of the 1980s.