M o t o r v a t o r V e r s io n 3

Version 3 MotorVator Highlights   Jan 2007

There are a number of new instructions and functions in Version 3, which are supported by a new MeccCompiler III and a new AWOS Version 2.32 operating system. This is a summary of the new features.

Timer Events

The MotorVator now supports User Timer Events. The internal Clock of the MeccCode within the MotorVator runs at 128 Hertz, so one ClockTick = 1/128 of a second. Using the UserTimerEvents, you can set up for a routine to run in X ClockTicks.

When the number of ClockTicks is up, your routine will run immediately (once), regardless of what else is running.

There are two new related commands:

SetEvent now includes UserTimer1Event through UserTimer8Event.

SetTimer ?Timer#?,?ClockTicks? sets the Timer to trigger after X ClockTicks.

In the following example, we use two Events, linked to two of the new UserTimers (there are 8 user timers available). With one event, we turn on the display, and with the other we turn off the display. By having the “Turn On” Event start the timer for the “Turn Off” Event, and vice versa, we get a repeating sequence of on and off, but with a different on time and off time.

```
; example showing how to use the timer events
; to set up an alternate flashing sequence

;###############################################
; each time UserTimer1 goes off, turn the display on
Declare EventHandler DisplayOn()

DisplayChars "O","O"

; then set up Timer2 so it will turn off in 1/2 second
SetTimer 2,64

End eventhandler

;###############################################
; each time UserTimer2 goes off, turn the display off

Declare EventHandler DisplayOff()

DisplayChars " "," "

; and then set up Timer1 so the display will turn on in 2 seconds
SetTimer 1,256

End eventhandler

;################################################

Begin Program

Disable Events

SetEvent UserTimer1Event, DisplayOn
SetEvent UserTimer2Event, DisplayOff

Enable Events

SetTimer 1,1      ; start off with Timer1 to fire immediately

Loop
      ; then do nothing else, other than let the timers run
End Loop

End Program
```

Note that using the TimerEvents is preferred to using a Wait statement, because nothing else in your code can happen during the wait period. By using a TimerEvent, other functions can be allowed to operate.

For example, if you have a critical Event on an input (Say an emergency stop limit switch), then even if the input changes during the Wait, the event won't occur until the wait time has finished. Try with the following example with a microswitch connected to OnOff3 Input, and see that even though you change the microswitch, the "o3" is not displayed until the next Beep sounds (i.e. when the Wait 500 has completed).

```
Declare EventHandler OnOff3change()
        DisplayChars "O","3"
End EventHandler

Begin Program

SetEvent OnOff3Event, OnOff3change
Enable Events

DisplayChars "A", "0"
Loop
        Beep 100,10
        Wait 500
End Loop

End Program
```

GetClock(), Wait

In order to get faster Stepper Motor functions (See next section), we have increased the frequency of the internal clocks. This means that any of your existing code that uses WAIT or GetClock() will need to be changed.

The GetClock() function now runs at 1024Hz (rather than the current 111Hz), while the "main clock" now runs at the slightly faster 128Hz (rather than 111Hz). So to get a Wait of 1 second, use "WAIT 128". See the note under TimerEvents, about the dangers of using Long Waits…..nothing else in your code will run during a Wait statement, so you might miss an important event.

Note that the GetClock function returns a Word value, so the maximum period that you can time in one go is 65535/1024 = 64 seconds.

You can use the GetClock() function to program an "in-line" wait that still lets your events happen during the waiting time. E.g. example to wait for 2 seconds between beeps:

You can use the GetClock() function to program an "in-line" wait that still lets your events happen during the waiting time. E.g. example to wait for 2 seconds between beeps:

```
Declare  Function WaitClock(tickstowait as word)
          Declare Word startclock
          Declare Word thisclock
          Declare Word clockdone
          startclock = GetClock()
          Do
                    thisclock = GetClock()
                    clockdone = thisclock - startclock
          Until clockdone > tickstowait
End Function

Begin Program
Loop
          Call WaitClock(2048)          ; 2 x 1024 = 2 seconds
          Beep 50,1
End Loop

End Program
```

Stepper Motor Routines

There are three new instructions supporting Stepper Motor Operation.
ConfigStepper
SetStepper
Stepper1Event / Stepper2Event

ConfigStepper ?StepperNo?,?MinWait?,?MaxWait?,?HoldCurrent?

?StepperNo?	Which stepper: Stepper 1 uses Motors A&B, Stepper 2 uses Motors C&D
?MinWait?	How long to wait between steps, at the maximum speed.
?MaxWait?	How long to wait between steps, at the minimum (startup) speed
?HoldCurrent?	What percentage of full power to apply when the stepper is stationary to have it hold its position

Use these parameters to tune your stepper for maximum performance, without losing steps. Set the MinWait and MaxWait to the smallest level that does not cause you to lose steps. This will vary for each different type of stepper motor and voltage level.

SetStepper ?StepperNo?,?Direction?,?NumberOfSteps?

?StepperNo?	Which stepper: Stepper 1 uses Motors A&B, Stepper 2 uses Motors C&D
?Direction?	Forward or Backwards
?NumberOf Steps?	

Having issued a SetStepper command, the MotorVator will take over and handle the StepperMotor to turn the required number of steps, and will handle the acceleration and deceleration, based on the MaxWait and MinWait parameters of the ConfigStepper command.

Stepper1Event

So that you know that the Stepper has finished moving (because you don't want your code to have to wait until it has finished before executing the next instruction), the Stepper1Event (or Stepper2Event) will fire when the Stepper completes the NumberOfSteps requested in the SetStepper command.

Debug

One of the most powerful abilities of the MotorVator is that of being able to "debug" your code interactively, and step and trace each statement. For larger programs however, it can be tedious stepping through hundreds of statements to get to the area where you wish to look more closely. The DEBUG statement now allows you to set a "BreakPoint" within your code. In the Debugger, you can tell the code to run until it gets to the next Debug Statement, and then stop in the Debug mode. You can then use any of the debug tools (Single step, multiple step, jump, change data etc) to analyse your program's operation.

Unless you are in the Debug mode, the MotorVator will ignore the Debug statement, so it is safe to leave it in your code.

SetMotor

SetMotor is not a new command, but you can now use it with Variables for Motor Number, Direction and Speed, rather than the previous restriction that the Motor Number and Direction had to be fixed constants.

This makes it much easier to code common motor functions into subroutines without the need for unwieldy Select or If-Then-Else structures.

StopMotor

We’ve also added a StopMotor command that accepts a Motor Number, to go with the StopMotors that stops all motors.

So here’s the full range of motor commands:

```
Begin Program

Declare Byte   motorno = 1
Declare Byte   motordir ="F"
Declare Byte   motorspeed = 60

StopMotor 2                        ; stop Motor 2
StopMotor MotorNo                  ; stop Motor (1 at this stage)

StopMotors                         ; stop all motors

SetMotor 2,"B",50

SetMotor motorno, motordir, motorspeed


End Program
```

Receive and Transmit

Functions have been implemented that allow you to Send and Receive Data via the Serial port.

The data is sent and received in “Ascii Triplets” where three Ascii numerals define one byte value, e.g. “048” represents Hex Byte 030H (“0”). Each transmission is prefixed with ”073” (representing Ascii “I”) so that the CLI can ignore it.

This use of the “Ascii Triplets” is to avoid issues with sending non-printable control characters over the serial interface which has to be shared with the CLI.

To send data from a MotorVator, use the TRANSMIT command.

Transmit ?TransmitBufferPointer?

Where TransmitBufferPointer is a DataPointer to a Datatable of minimum 8 bytes in size. E.g.

```
Declare Datatable   TxBuffer
            “TX123456”
End DataTable
Declare DataPointer TXBuffPointer

Begin Program

TXBuffPointer = SetPointer(TxBuffer)

Transmit TxBuffPointer
………..
```

This would result in the following being sent out the serial point
073084088048049050051052053<CR>

To send data to a MotorVator, the sender needs to prefix the data string with “I”, and then send up to 8 triplets, followed by a single Carriage Return (CR – code 13).

Once the CR is received, the MotorVator will raise the ReceiveEvent. Users need to provide an EventHandler for this event.

Once the ReceiveEvent is raised, the user can read in the received data using the RECEIVE command.

Receive ?ReceiveBufferPointer?

Where ReceiveBufferPointer is a DataPointer to a Datatable of minimum 8 bytes

The Ascii Triplets are converted back into Byte values and stored into the 8 bytes pointed to by ReceiveBufferPointer.

Note that if less than 8 Triplets are sent before the Carriage Return, then the rest of the 8 bytes will be set to 00. e.g. sending I001002003<CR> will give 01 02 03 00 00 00 00 00 in the receive buffer datatable.

Here is an example program

```
; test of send and receive

; to use - Set the program running from the CLI, using the "G" command
; press the I key, then press 0 0 3 and CR (Enter)
; it will display 073003000000000000000000
; every 5 seconds it will display 073084088032021082088032001 or similar


; set up a transmit buffer that will have
; text TX followed by count and text RX followed by count
;e.g. TX01RX01
Declare         DataTable       txbuffer
```

```
          "TX "
End DataTable

Declare Byte TXCount  = 0

Declare DataTable               txbugRX
          "RX "
End DataTable

Declare Byte RXCount = 0

Declare         DataTable       rxbuffer
                Reserve 8
End DataTable

Declare         Byte            RXTemp
Declare         DataPointer     TxBuffPointer
Declare         DataPointer     RxBuffPointer


Begin Program

SetEvent        ReceiveEvent,RxEvent
SetEvent        TransmitEvent,TxEvent
SetEvent        UserTimer1Event, txtimer
Enable Events

TXBuffPointer = SetPointer(TxBuffer)
RXBuffPointer = SetPointer(RxBuffer)

SetTimer 1,1    ; set the timer so it will send the first message

Loop
; do nothing and let the events do the receiving and sending
End Loop

End Program


Declare EventHandler txtimer()
          ; transmit the buffer
          TXBuffPointer = SetPointer(TxBuffer)
          Transmit TxBuffPointer                ; send the transmit
          SetTimer 1,640        ; and set up to send again in 5 seconds (5 x 128 ticks)
End EventHandler

Declare EventHandler rxevent()
          ; each time we get a Receive event, increment the RX Counter
          ; that will go out in the Transmit
          Increment RXCount
```

```
        Beep 50,1
        ; and read in the incoming data
        RXBuffPointer = SetPointer(rxbuffer)
        Receive RXBuffPointer
        ; display the first byte on the MV display
        RXBuffPointer = SetPointer(rxbuffer)
        RXTemp = ReadData(RXBuffPointer)
        DisplayHex RXTemp
        ; and send the whole buffer out to display
        txbuffpointer = SetPointer(RXBUFFER)
        Transmit TxBuffPointer
End EventHandler

Declare EventHandler TXEvent()
        ; each time we finish sending something, increment the counter
        ; just to show how the Transmit event works
        Increment TXCount
End EventHandler
```

Note that you must not try to transmit if there is already a transmission in progress. This is why the TransmitEvent is provided, to indicate when the last transmission has completed.