meccanisms

Powerful and Painless Electronics for Meccano Modellers

MotorVator 4 User Manual

Release 3.00
Dec 2006
For AWOS 2.31 onwards and MeccCompiler III

Release Date	Notes
December 2006	Includes sections on new instructions.

Meccanisms – Bringing Models to Life

Meccanisms Limited markets a range of intelligent control equipment and accessories, aimed at the hobby and educational markets.

Meccanisms Limited
P O Box 26-179
Epsom
Auckland 1003
New Zealand

www.meccanisms.com
info@meccanisms.com

The MotorVator 4

With the Meccanisms Motorvator 4, you have purchased a powerful and flexible microprocessor controlled device. With the capacity to simultaneously process input from fourteen sensors and control nine devices, the MotorVator will form the basis for as many intelligent machines as your imagination can produce.

MotorVator 4 capabilities:

- Pulse Width Modulated Control of four DC motors,
- Split drive voltages to allow you to mix large and small motors.
- Six Analogue Voltage Sensing Inputs
- Four Timed (Pulse Width Measurement) Inputs
- Four Digital Inputs
- Two Servo Motor Control Outputs
- Three Controlled Action Outputs.

Table of Contents

Safety Information

Before operating the MotorVator, please read the complete manual thoroughly.

Do not attempt to disassemble the unit, other than expressly described in this manual. There are no user-serviceable items within the case.

Stop operating the unit immediately should it emit smoke or fumes. Disconnect the unit from any external power supply. Disconnect the battery. Contact your service agent.

Do not operate the equipment if it has been dropped or appears physically damaged. Contact your service agent.

Prevent the unit from coming into contact with water or other liquids. If the unit is wet, do not operate. Disconnect the battery and leave the battery cover off. Dry the exterior and place in a warm (max 30ºC) dry place and allow to dry out before attempting to operate.

Do not short-circuit any of the terminals, other than expressly described in this manual.

Do not supply more than 15 Volts to any of the Power Input Sockets.

Do not connect anything other than a 5V input to any input port.

Do not connect other than a standard 9V battery to the battery clip.

Do not exceed the rated current (amperage) for any port. Specifically, do not exceed 300mA for the Action Outputs or 1A continuous on the Motor Outputs.

Remove the battery before storing for extended periods

The MotorVator is a powerful, flexible device, able to control a wide range of external items.

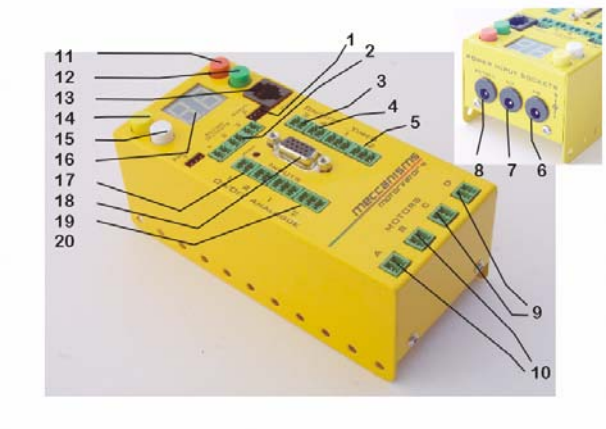
We have no control over how you use the MotorVator, the programs that you load into the MotorVator, what you connect to it, or what the resulting combination does.

Meccanisms Limited takes no responsibility for any harm, damage or other claim caused by any 'machine' controlled all or partly by the MotorVator.

We strongly recommend taking a minimum of the following precautions when constructing a controlled machine:

- ❑ Always test your machine thoroughly. Before powering the machine with motors under MotorVator control, test each movement manually.
- ❑ If possible, run initial tests using low capacity batteries or a "current-limited" power supply. Any faults or shorts in wiring will show up without causing major damage or danger.
- ❑ Test the operation of motors at a slow speed initially, then increase to normal operating speed.
- ❑ Provide safety switches, end detection sensors, clutches etc to ensure that any movement outside of the expected is detected and controlled.
- ❑ Provide an emergency stop facility that is clearly labelled and located in an obvious and non-obstructed position. Assume that this could be pressed at any time, by any observer, so ensure that shutdown is graceful and safe. For example, if your model crane is carry a heavy load, make sure that an Emergency stop does not simply drop the load

MotorVator Layout Diagram



Basic Operation MotorVator 4

Getting Started

Running Preprogrammed Modes

The MotorVator comes with a number of preprogrammed Modes, that will allow you to control many different types of models without any need for programming. We will discuss these first, then go onto the User Programming.

a) Unpack the MotorVator and the Director.

b) Connect the Director cable to the MotorVator Sensor Expansion Port (Layout Diagram, 18).

c) Obtain a suitable Power Source. It should provide a minimum of 7.5Volts DC and have a 2.1mm DC Power Plug wired as per the diagram on page 21……

	IT IS VITAL THAT THE PLUG IS WIRED CORRECTLY. If not, you will Damage the MotorVator.

Sources of suitable Power Supplies include

a. Batteries: Dry Cell or GelCell batteries. For free-running models (e.g. robots) the Radio Controlled Hobby has large capacity Rechargeable battery packs that are ideally suited.

	Be aware however that these batteries can provide a very high current for a short period of time. It is recommended that you should test your models using standard dry cell (e.g. Eveready) batteries first.

Note that the C-Tick Approval (Z204) for the

MotorVator has been issued for battery use only. This means that there is has been no test completed to confirm that the Motorvator does not exceed Electro-Magnetic Radiation (in laymans' terms "interference with radio receivers") under operation with a mains-powered external power supply.

b. DC Power supplies (e.g. Train Controllers, Power Packs etc). Make sure that they are DC and not AC – many cheap power packs are AC. Be careful about using cheap "plug packs", even if they are DC. Many of these are unregulated which means that they will reduce voltage from mains supply down to the nominal stated voltage, but they do not get rid of the ripples, nor will they give out a constant voltage under load. Only use a plug pack that is Regulated.

c. The power supplies from Old Personal Computers. These typically supply clean power at 5V and 12V, with high current capacities. Leave them in their PC cases for safety, and connect to the big connectors that went to the Disk Drives. You will find both 5V and 12V on these connectors.

Make sure that the current capacity from the power supply is greater than the sum of all the outputs that you want to operate concurrently. Each motor may require upwards of 1 Amp. Any Servo Motors (see page 40) will require 0.5A while turning, and if you want to use the Action Ports (see page 39) then you need to allow for the consumption on these ports also.

d) Connect the Power Source to the Motor AB Power Input (Layout Diagram, 6). Turn the Power On.

e) The MotorVator should play a short fanfare, then display "ON" on the two character display.

f) The four keys on the MotorVator work in a consistent manner:

Key	Main Functions
RED (OFF) Layout Diagram, 11	- Stops Programs Running - Hold Down for two seconds to Turn MotorVator Off.
GREEN (ON) Layout Diagram, 12	- Start Program - Turn MotorVator On.
Yellow (UP) Layout Diagram, 13	- Change Selection
White (Down) Layout Diagram, 14	- Change Selection

Basic Operation:

I. MotorVator Displays ON

II. Press UP/Down to change the Program Number (Cycles from 00 to 99). Program Mode 99 is a special mode and is used for a Program created by the User on the MeccCompiler Software Suite and downloaded to the MotorVator.

III. Press ON (Green) to go to Select Program

The MotorVator will beep three times, then start running the program. This is to give you time to be ready with your model.

Each Program Mode uses a different combination of motors, Director controls etc. See page 45 for tables showing which Director controls attach to which motors, servos etc.

IV. Press OFF (Red) to stop the program.

V. To test, plug a motor or motors into MOTOR Outputs A through D.
Select Program 01. Press GO (Green).
Move the joysticks on the DIRECTOR and the motors should run.

VI. Holding down the OFF (Red) button for two seconds when a program is not running will turn off the MotorVator. Press the ON (Green) button to start up again.

Backup Battery

If all you ever want to do is run the Pre-Programmed Modes, then there is no need to fit the Backup Battery.

If however you want to write your own MeccCode programs, download them to the MotorVator and have your program available in the MotorVator at another time without having to download it again (e.g. you want to program the MotorVator at home then take it to a meeting or exhibition), then you need to fit the Backup Battery which will save your data between sessions.

To fit the battery, remove the two screws (marked below) that hold the Battery Compartment cover on the underside of the case. Connect the Battery Snap Lead to the 9V Battery, making sure of the correct polarity of the connector. Press the battery into the holder and the re-fit with the screws. Some 9V batteries are larger than others, and they may cause pressure on the circuit boards within the MotorVator. We recommend fitting 2 standard Meccano washers between the battery cover and the Motorvator case, to provide some additional clearance.

Note to ensure that the memory contents of the MotorVator are maintained, change the battery while the unit is powered by an external source, and always shut the MotorVator down with the OFF key before removing external power.

Use the H (Help) command of the CLI to read the current battery voltage. Change the Battery when the voltage falls to 6V.

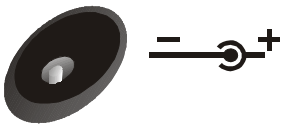
Ports

The MotorVator supports a wide range of inputs and outputs.

DC Voltage Inputs

There are three 2.1mm DC standard power sockets (Layout Diagram, no 6, 7 and 8). Use these to supply DC power for the various devices being controlled.

NOTE: THESE CONNECTORS MUST BE WIRED CORRECTLY – THE GROUND GOES TO THE PIN IN THE MIDDLE.

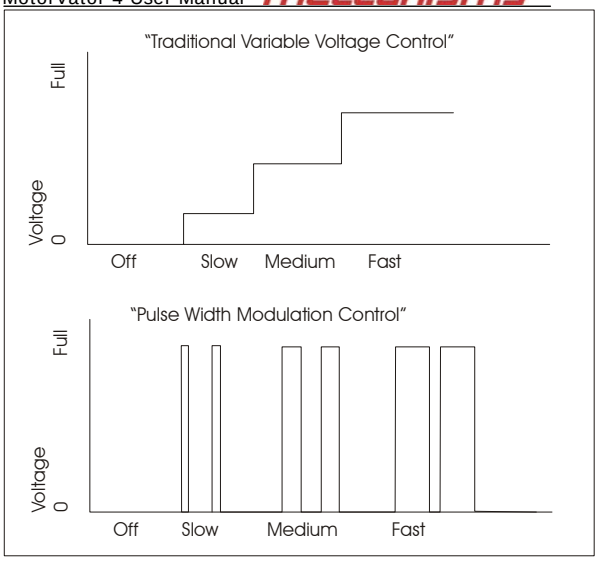


Connecting the Positive and Negative incorrectly WILL DAMAGE THE UNIT.

Motor AB Power Socket (Layout Diagram, 6)

Power connected to this socket is used to drive the Motor Outputs A and B (Layout Diagram, no 10). The motor switching circuits will provide Pulse Width Modulated (PWM) signals to the DC motors connected to A and B using this voltage source. The Amplitude (maximum voltage) of the signals from connectors A and B will match the input voltage to this socket.

Pulse Width Modulation is a way of controlling speed by controlling the width of full-voltage pulses rather than controlling the voltage level itself. It has benefits of providing smoother speed control, plus good low speed operation because the use of the full voltage at all speeds overcomes resistance issues that hamper operation at low voltages.



Note that if you do not supply a different voltage to the other two DC Voltage Input sockets, then the input voltage to the Motor AB Power Socket is used for all the devices.

If you only provide one power source, then it must be a minimum of 6 Volts. We recommend using a single 9V or 12V switch mode supply for most applications.

Motor CD Power Socket (Layout Diagram, 7)

If you connect a power source to this socket, then two things happen:

1. Power connected to this socket is used to drive the Motor Outputs C and D (Layout Diagram, 9), and used for the Action Ports (if nothing is plugged into Action Power Socket (Layout Diagram, no 8).
2. The Power connected to Motor AB Power Socket is now only used for Motors A and B.

Note also that if you plug a power source into this Motor CD Power Socket and don't plug anything into the Motor AB Power Socket, then no voltage is available for Motors A or B.

Note that regardless of the voltage connected to the Motor AB socket, if you connect power to the Motor CD Socket and do not connect voltage to the Action Input Socket, then the Motor CD Voltage supply must be a minimum of 6 Volts.

Action Power Socket (Layout Diagram, 8)

If you connect a power source to this socket, then two things happen:

1. Power connected to this socket is used to drive the Action Outputs (Layout Diagram, 2) and the Servo Outputs (Layout Diagram, 1), and used for the internal workings of the MotorVator (and for any sensors that require power).

2. The Power connected to Motor AB Power Socket (#6) (and possibly Motor CD Power Socket #7) is now only used for the Motor Outputs.

Note also that if you plug a power source into this Action Power Socket and don't plug anything into the Motor AB Power Socket or Motor CD Power Socket, then no voltage is available for Motors A or B or C or D.

The power into the Action Power Socket is also used to create the 5V supply needed to operate the Servo Motors (ports 1), so it must be a minimum of 6Volts.

If you want to run motors with a voltage of less than 5 Volts, then

1. Connect the lower supply voltage (e.g. 3 Volts) to the Motor AB and/or CD Input Sockets.
2. Connect a minimum 6V supply to the Action Input Socket.

You can operate two different motor voltages, plus a third voltage for the Action Ports. For example:

Input Socket	Voltage Supplied	Voltage Available
Motor AB -	3V	3Volts for Motors A and B
Motor CD	12V	12V for Motors C and D
Action	6V	6V on Action Outputs 5V on Servo Outputs 6V to run MotorVator

In most applications, a single 9-12V supply into the Motor AB Input Socket will be suitable.

Communications Port (Layout Diagram, 13)

The communications port allows you to connect to a Personal Computer and

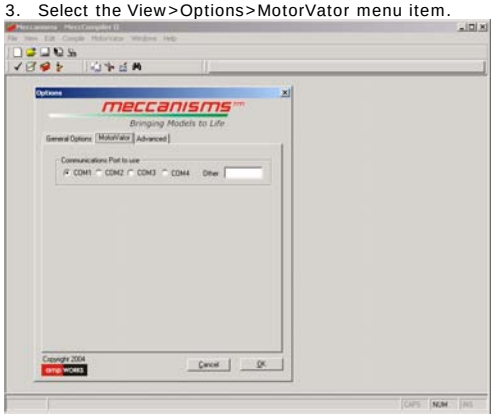
- Test individual ports using the CLI (see below).
- Download User Programs from CodeWriter and MeccCompiler.
- Debug your MeccCode Program.

Command Line Interface (CLI)

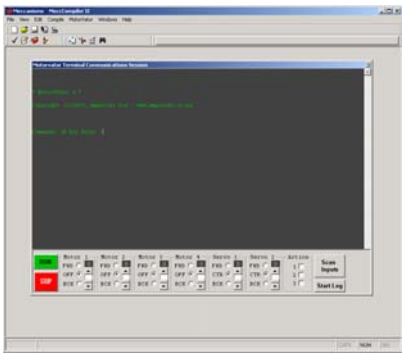
The Command Line Interface (CLI) allows you to instruct the MotorVator to perform a single function. This is useful when testing your model during construction.

To use the CLI, you need to

1. Connect the MeccCode Interface Cable to a serial port on your PC.
2. Install and Run the MeccCode CompilerII (on the included CD).
3. Select the View>Options>MotorVator menu item.



4. Select the Communications Port that you are connected to. If you are using a COM port other than COM1 through COM4, then enter the name (e.g. COM5) in the Other Field.
5. Now Select the MotorVator>Open Terminal Window Option



6. When you start up the MotorVator you should see "Enter Command >" in the Communications Window.
7. You can use the "Radio Buttons" section at the bottom of the Communications Window to manually test your hardware.
8. Experiment with the CLI Commands. Enter Pa,b,c and you will hear a short tune.

9. The valid CLI Commands are

Command	Format	Explanation
Action Port	Aps	Set the action Ports P is port number (1,2 or 3) S is state, (0 or 1) e.g. A11 will turn Action Port 1 on. A30 will turn Action Port 3 off.
Clear User Program Memory	C	Will Clear (Zero) out all the User Program Space. Use if your program has had problems and potentially overwritten user space, before reloading.
Debug User Program	D	Enters a debug mode for your user program. Allows you to "Single Step" your program (i.e. execute one statement then stop). To use, Enter "D" Will display the Instruction just executed, plus the relevant variables. This display will be in numeric only format (use the OpCode table to decode). Press Enter to execute next instruction G – stop debugging and let program Go from next instruction. Mxxxx – displays 64 bytes of data starting from address xxxx. User Program space starts at address 0000. Aaaaadd – allows a single byte at address aaaa to be changed to value dd. Both aaaa and dd are in Hexadecimal format. R – restart your program from address 0 J – Jaaaa causes your program to execute from address aaaa. X – exits debug mode and stops

Command	Format	Explanation
Run Program (go)	Gpp	pp is two digit program number. 99 is for the User Program. Any other number will run a Preprogrammed mode.
Help	H	Displays the version number, serial number, Battery Voltage and a list of Valid CLI commands.
Set Stepper variables	J	Sets the three variables used by the Stepper Motor Commands n – 1 or 2 for Stepper 1 (MotorAB) or Stepper 2 (MotorCD) mmm – three digit minimum wait between steps. Set to the smallest number that allows your stepper to move reliably without missing steps xxx- 3 digit maximum wait between steps. Sets the initial step wait (when ramping up to full speed) hh – 2 digit hold percentage current, applied when the stepper is not moving, to hold its position. Typically use 10% e.g. J101005010
Activate Stepper Motor	K	Move Stepper motor n = 1 digit Stepper # (1 or 2) d = Direction (F or B), SSSS = steps (maximum 2559) e.g. K1F1234
Load User Program	L...	Load in a MeccCode program (in object format). Load command expects lines of decimal values, separated by commas, with the final line ended with a "]" character.
Motor	Mmdsss	m – 1,2,3 or 4 for the required motor. d – F or R for forward or reverse s – 000 to 100 as a percentage of full power. e.g. M1F050 will start Motor 1 in the Forward direction at 50% duty cycle. Use this command to test which direction the MotorVator thinks is forward and

		backwards, and to test maximum and minimum speed gearing. If the motor turns the "wrong way", you can simply swap the wires on the connector at the MOTOR socket.
Phone Tune	P.....	Play an RTTTL Format tune. RTTTL is RingToneTransportableLanguage and has the following syntax: P [<duration>]<Note>[<sharp>][.][<octave>][,...] Duration is 1,2,4,8,16 for full, half, qtr, eight and sixteenth beats Note is A to G, P for No Sound Pause. Sharp is "#" and optional. "." means add half the duration again to the note, Optional. Octave is 5 or 6 and is optional. Examples: **1c,1d,1e** will play equal three notes in rising succession. The fanfare when MotorVator starts up is 16c,16e,16g,c6,16g,2c6. Extension for MotorVator – if the last character is a "<" then the tune will repeat continuously (until another tune is loaded or a Tone is played). You should be able to download a variety of tunes from the internet, by searching for RTTTL. Note that the MotorVator does not want spaces between the notes.
Read Status	R	This will give a repeat reading of the input ports (On/Off, Timed and Analogue), on a one second update. The values displayed will be the same as you can use within a program, hence it's a good way to check out your model. For example, if you are using a potentiometer to measure the angle of a robot arm, move the arm manually (either by hand or using the Director controls) while noting the readings that come from the Analogue port to which you have connected your potentiometer. Note that

		the Read Status will work even while you're running a Program. Press ENTER to cancel out of the status read.
Servo	Ssdaa	s – 1 or 2 for the required servo d – direction, B or F aa – angle 00 to 90 degrees e.g. S1B45 will turn Servo #1 Back (anticlockwise) 45 degrees from center.
Tone	Ttttdd	Play a single Tone ttt is a numeric value related to pitch (larger numbers give lower tones) and dd is the duration in 1/10 of a second e.g. T10010 will play a 1 second tone.
Stop all Motors etc	Z	Stops all motors, and resets Servos to centre position.

On/Off Inputs (Layout Diagram, 3,4,17)

These four inputs are the easiest way to get information into the MotorVator and your programs. Each of these inputs measures either an On (Closed) or Off (Open) state. The simplest way to use them is to connect the two pins on the connector together with a switch. This switch can be microswitch, a button switch or similar. E.g. Use this switch for simple 'bumpers' in a basic robot.

Note the polarity (which way round) the plug goes when connecting to other equipment. If you're connecting a microswitch the polarity doesn't matter.

In the MeccCode Software the ?variable? = READONOFF(x) instruction is used to get the current state (0 for Open, 1 for Closed) of each of the On/Off Inputs.

There is also a ?variable? = READPULSECOUNTDIGITAL(x) command that will return the number of pulses since the last time you asked. See Page 180.

TECHNICAL NOTE:
These input ports measure a TTL logic level signal. You can therefore connect more sophisticated TTL logic level circuits to the Signal Line. These inputs will show an Off if the Signal pin is High (5V) or an ON if the input pin is Low (0V).

Because these input ports have an internal pull-up resistor, you can use the chassis of your Meccano model as a "Common Earth" and only have one wire from the microswitch back to the MotorVator, with a connection from the chassis to one of the GND pins on at least one of the Input Ports. The other Microswitch connector goes to the chassis to complete the circuit.

DO NOT CONNECT any signal that exceeds 5 Volts to the signal pin.

E.g. in this example, we check to see whether a bumper switch on the front of our robot is closed (e.g. have we run into something?). If it is, then we will reverse.

```
Switch= readonoff(1)
If switch <> 0 then
      SetMotor 1,"B",speed
End if
……
```


Timed Inputs (Layout Diagram, 5)

These two input ports allow you to get input from a 0-5V logic level device that provides both the current state of the input, plus additional information about the length of the last pulse into these ports.

The Three pole connector on these ports provides +5V, Ground and the Signal_In line. Use the +5V to power compatible sensors.

⚠	**DO NOT CONNECT any signal that exceeds 5 Volts to the signal pin.**

Within the MeccCode Software, the
?variable? = READTIMEDSTATE(x)
instruction (see page 181) stores the current status into a variable that shows whether the port is currently High or Low (so you can use these Timed Input ports as simple on/off input ports)

There is also another instruction
MyByte = ReadTimedPulseWidth(x)
that gives the width of the last incoming pulse. Use this latter variable for calculations on distance, temperature etc depending on the sensors characteristics.

There is also a

?ByteVarName? = **ReadPulseCountDigital(***?TimedPortNo?***)**

Instruction that will return the number of pulses since the last time you asked. See Page 180.

NOTE:
1/ There are a further two Timed Input ports available through the Sensor Expansion Connector (Layout Diagram, 18).
2/ The Timed Inputs are shared with the Director Buttons.

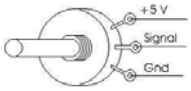
E.g. in this example, we read the value from a Distance sensor (connected to TIMED INPUT 1), and decide whether we have got too close to the object….

```
distance = ReadTimedPulseWidth(1)
if distance < too_close then
      StopMotors
End if
….
```


Analogue Inputs (Layout Diagram, 19)

These two ports allow you to measure variable voltage levels (in the range 0 to 5V DC).

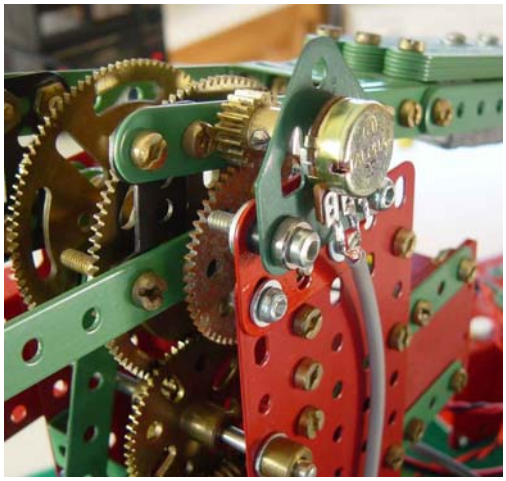
The Three pole connector on these ports provides +5V, Ground and the Signal_In line. Use the +5V to power compatible sensors, and to provide the reference voltages for the potentiometer or your own sensing circuits.



The simplest way to use these ports is with a Potentiometer. When wired as shown above, turning the knob on the potentiometer (which is just like the volume knob on a radio) will change voltage on the Signal line between 0 and 5 Volts.

⚠	**ONLY USE THE 0 and 5V reference voltages supplied at the MotorVator port.**

The following photo shows a potentiometer connected via a 3:1 gearing to the arm of a robot.



As the robot arm moves through its full range (about 80 degrees), the potentiometer is geared to move through most of its full 270 degree range and so provides a varying analogue voltage signal between 0 Volts at one end of the range (Analog value = 0) and 5 Volts (Analog value = 255) at the other end of the range.

In the MeccCode software, use the
?ByteVarName? = **ReadAnalogue(***?AnaloguePort?***)**
instruction command to get the current value.

For example, the following routine will start the Robot Arm moving upwards, and wait until it gets to a predefined position, then stop the arm:

```
SetMOtor    ArmMotor,"F",armspeed
Do
      Armheight= ReadAnalogue(1)
Until Armheight > Arm_Top
StopMotors
```

There is a wide range of sensors that provide information in this format. The Meccanisms IR Distance Sensor is just one example, where the voltage reading from the sensor is related to the distance of the detected object.

There are an additional Four Analogue Inputs available through the Sensor Expansion Connector (Layout Diagram, 18). These can be accessed using the Analogue Expansion Module (see page 55), or the optional Sensor Expansion Cable.

Action Outputs (Layout Diagram, 2)

There are three Action Output Ports. Each of these ports can be turned On and Off from within your programs. When the Port is On, it provides Voltage at a maximum current of 300mA (per port).

DO NOT CONNECT THINGS THAT WILL DRAW MORE THAN THIS CURRENT. YOU WILL BURN OUT THE TRANSISTORS. Shorting the pins will do the same. If you want to control high current items, then use the Motor Output Ports or connect a relay or external transistor circuit to the Action Output.

You can use this output to power additional [small] motors, external lighting, solenoids etc. Note that any motor connected to these Action Output will be either Off or Fully On – unlike the Motor Ports there is no speed control or reversing capability.

Note that the +Voltage PIN is always Live, therefore care needs to be taken not to short the +Voltage Pin to ground even if the Port is turned OFF.

The power going to these outputs comes from the Action Power Source Connector (Layout Diagram, 8, see page 24).

Technical Note:

The drive current for the Action Output Ports is provided by pulling the Ground Pin Low when the Port is turned ON. When the Port is Off the Ground Pin floats to 1.2V below the Action Output Voltage. The +Voltage Pin is always at the Action Output Voltage.

Use the
SetAction ?PortNo?, ?ByteValue?
Instruction to set Action Port (PortNo) to On (1) or Off(2)

DO NOT Exceed a current drain of 300mA. A higher current will burn out the internal transistors.

Servo Motor Control Outputs (Layout Diagram, 1)

There are two ports dedicated to controlling Servo Motors.

There are standard software functions within the MeccCode software so you can control the Servo motors to a specific angle. Several of the PreProgrammed Modes also control the servos.

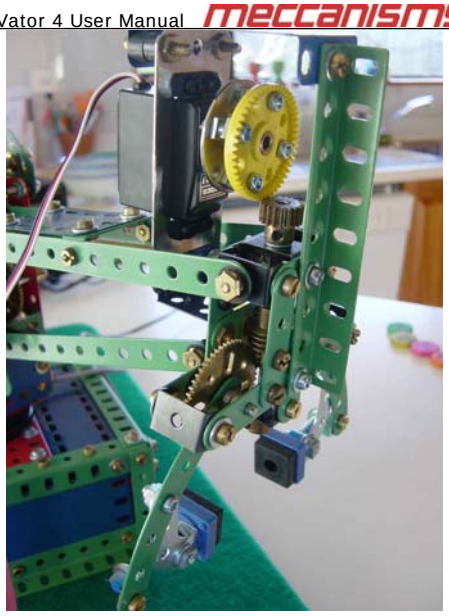
There are two 3 pin connectors that will take a standard Futaba/Hitec servo cable directly.

Use these servo motors for steering, mount a sensor on a servo and create a 'radar scan sensor', use two servo motors to create a walking motion, have your robot turn a door handle through ninety degrees etc.

Servo Motors:

Servo Motors are a special kind of DC motor that do not simply turn over and over when voltage is applied. A Servo Motor is designed to rotate a programmed amount and then hold that position. They are typically used in Radio Controlled models (cars, boats, planes etc) to control rudders, steering, throttle settings etc. The amount of rotation is determined by the width of a pulse that is send to the Servo. The Servo will maintain this position until a different pulse is sent.

The following photo shows a Servo Motor connected to operate the grab on a robot arm, where by turning through 90 Degrees Left to 90 Degrees Right, the jaws of the grab close and open.



Optional Accessories and Modules

Meccanisms Director

The Meccanisms Director connects to the Sensor Expansion Port and provides two XY Joysticks and Four Control Buttons, to give you manual control over the most complex models, particularly cranes.

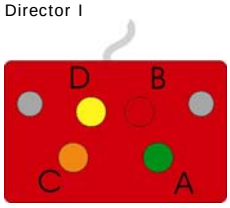
Within the MeccCode Software are instructions that give you access to the values of the Joysticks and Buttons.

Instruction	Number		
CALIBRATE JOYSTICK	Joystick 1 or 2	DeadBand X	Deadband Y
READ JOYSTICK1	X or Y	Joystick direction (from center)	Joystick Reading
READ JOYSTICK2	X or Y	Joystick direction (from center)	Joystick Reading
READ BUTTON	Button 1..4	Closed or Open	

The MotorVator has a number of Standard PreProgrammed Modes that “Connect” the joysticks and buttons on the Director to the Motors, Action Ports and Servo Ports on the MotorVator.

The following tables show the available modes. The button layout on the two models of Director is as follows:

Director I



Director II

This page left intentionally blank.

Mode 1

4 motors, with momentary action on Servos and Action ports

	Motor 1	Motor 2	Motor 3	Motor 4	Servo 1	Servo 2	Action 1	Action 2	Action 3
Joystick 1 X	X								
Joystick 1 Y		X							
Joystick 2 X			X						
Joystick 2 Y				X					
Button A – Pressed					F		On		
Button A - Released					B		Off		
Button B - Pressed						F		On	
Button B - Released						B		Off	
Button C - Pressed									
Button C - Released									
Button D - Pressed									On
Button D - Released									Off

Servos stay in Full AntiClockwise (Left) position unless Buttons are Pressed and Held down.

Mode 2

4 motors, with Flip-Flop action on Servos and Action ports

	Motor 1	Motor 2	Motor 3	Motor 4	Servo 1	Servo 2	Action 1	Action 2	Action 3
Joystick 1 X	X								
Joystick 1 Y		X							
Joystick 2 X			X						
Joystick 2 Y				X					
Button A – Pressed					F		On		
Button A - Released									
Button B - Pressed					B		Off		
Button B - Released									
Button C - Pressed						F		On	
Button C - Released									
Button D - Pressed						B		Off	
Button D - Released									

Notes: Servos move to position when Button Pressed and stay in that position until the other button is pressed. Good for operating Crane Buckets, Grabs. Use the Action Ports for an ElectroMagnet grab on a scrap crane.

Mode 3

2 motors, 2 joysticks to two servos. Flip-flop on Action ports

	Motor 1	Motor 2	Motor 3	Motor 4	Servo 1	Servo 2	Action 1	Action 2	Action 3
Joystick 1 X	X								
Joystick 1 Y		X							
Joystick 2 X					Proportional				
Joystick 2 Y						Proportional			
Button A – Pressed							On		
Button A - Released									
Button B - Pressed							Off		
Button B - Released									
Button C - Pressed								On	
Button C - Released									
Button D - Pressed								Off	
Button D - Released									

Notes – One Joystick controls 2 motors, the other 2 servos. Use this mode for a simple 1 Motor (driving) and 1 Servo (Steering) model.

Mode 4

2 motors, 2 joysticks to two servos. Flip-flop on Action ports

	Motor 1	Motor 2	Motor 3	Motor 4	Servo 1	Servo 2	Action 1	Action 2	Action 3
Joystick 1 X					Proportional				
Joystick 1 Y	X								
Joystick 2 X						Proportional			
Joystick 2 Y		X							
Button A – Pressed							On		
Button A - Released									
Button B - Pressed							Off		
Button B - Released									
Button C - Pressed								On	
Button C - Released									
Button D - Pressed								Off	
Button D - Released									

Note – this is the same as Mode 3, but the Motors and Servos are separated so that each joystick has One Motor and One Servo.

Mode 5

3 motors, 1 servo. Flip-flop on Action ports

	Motor 1	Motor 2	Motor 3	Motor 4	Servo 1	Servo 2	Action 1	Action 2	Action 3
Joystick 1 X	X				Proportional				
Joystick 1 Y		X							
Joystick 2 X			X						
Joystick 2 Y							On		
Button A – Pressed									
Button A - Released							Off		
Button B - Pressed									
Button B - Released								On	
Button C - Pressed									
Button C - Released								Off	
Button D - Pressed									
Button D - Released									

Notes – Similar to 3 and 4, but with Three Motors and One Servo.

Mode 6

4 motors, with Flip-Flop action on Servos and Action ports

	Motor 1	Motor 2	Motor 3	Motor 4	Servo 1	Servo 2	Action 1	Action 2	Action 3
Joystick 1 X	X								
Joystick 1 Y		X							
Joystick 2 X			X						
Joystick 2 Y				X					
Button A – Pressed					B		On		
Button A - Released					Centre				
Button B - Pressed					F		Off		
Button B - Released					Centre				
Button C - Pressed						B		On	
Button C - Released						Centre			
Button D - Pressed						F		Off	
Button D - Released						Centre			

Notes – Servos rest at Centre Position, but move to either Left or Right while buttons are held down.
Action Ports turn on while button pressed

This page left intentionally blank.

InfraRed Distance Sensor

The Meccanisms InfraRed Distance Sensor is a Sharp IR Ranger model GP2D12, designed to give analogue readings for distances in the range 10cm to 80cm. The Meccanisms Sensor comes prewired for direct connection to an Analogue port.

This is the cheapest way of getting distance/proximity information into the MotorVator.

SW1 Bumper switch

The Meccanisms SW1 is a simple microswitch, mounted to make it easy to use with your Meccano model. It comes prewired for direct connection to an On/Off Port.

Servos Mounting Set

The Meccanisms Servo Mounting Set is designed to mount a Futaba S3003 (or compatible) Servo Motor within the Meccano Geometry. The Mounting Plate aligns the Servo Motor, and the Horn Plate mounts to the Servo Horn to give you the equivalent of a Bush Wheel on which to attach cranks or other gears (see picture page 40).

The Servo Mounting Set can be supplied with or without the actual servo motor.

Sensor Expansion Cable

The Sensor Expansion Port (Layout Diagram, 18) gives access to four additional Analogue Inputs and Two additional Timed inputs. The Optional Sensor Expansion Cable gives you access to these inputs.

The Sensor Expansion Port Connections (Shown looking down into the Female Port Connector) are:

D15

+5V			+5V
Timed Input 2			Timed Input 1
Analog 3			Analog 4
Gnd			Reset
Gnd			Analog 5
Analog 6			Timed Input 3
Timed Input 4			+5V
+5V			

D15 HD

+5V		Reset
+5V		Gnd
Timed 1		Analog 4
Analog 5		Timed 3
Timed 4		Analog 6

Middle Column, from Top
Timed 2
Analog 3
Gnd
n.c.
+5V

Technical Note:

These four additional analogue ports are shared with the Director Joystick inputs. Joystick One shares Analog ports 3 and 6 and Joystick Two shares Ports 4 and 5.

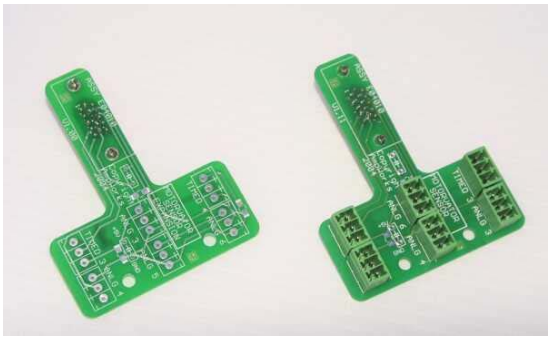
The Timed Inputs are used for the Director Buttons.

This information is provided to give you the maximum flexibility in using the MotorVator. Ampworks takes no responsibility for any damage done to, or caused by the MotorVator due to incorrect connections.

Analogue Expansion Modules

The Analogue Expansion modules gives you access to four more analogue ports. (see page 36), and two more Timed Inputs.

Each of the six connectors has the same three connectors – Ground (0V), 5V and Signal In. Use the Ground and 5V wires to power the sensor, or as reference voltages for your own circuits.



The MotorVator software has standard instructions for all six analogue ports (2 on the main MotorVator unit and these four on the Analogue Expansion module), which read the current voltage reading for each port.

More Accessories

For the Latest list of accessories, visit www.meccanisms.com and www.meccparts.com (see under Meccanisms>Accessories).

MeccCode II Software

The most powerful aspect of the MotorVator is its ability to be programmed to perform complicated sequences of actions. The programming of the MotorVator is done using the MeccCode language. The following sections explain the concepts and the available instructions.

Before we look at the MeccCode language, let's make sure that we're all on the same wavelength with the jargon, and look at some definitions and concepts.....

Definitions

Compiler

The program that converts human readable instructions (source or programming language) to machine language (object code).

Source Code

A human readable list containing instructions following a known format (the programming language syntax) that will be converted into Object Code by a Compiler.

Often call "the program", and written by a computer programmer.

Object Code

A list of Machine-readable values, that instruct the machine (in this case the MotorVator) to do certain things.

Syntax

The correct form of an instruction. Think of it as Grammar for computers. We tend to be able to understand English even if it is not writ proper. Computers are much fussier – to them it is either right or wrong.

Instructions

The instruction is the fundamental element in program preparation. Like a sentence, an instruction usually consists of a subject, an object and a verb.

The Verb is the “What?” – what are we going to do here. Each machine has a limited number of built-in operations that it is capable of executing. Also called the “Operator” or “Opcode” – short for Operation Code.

The subject and object are the things that we are doing the “What?” with, and are often called Operands, or Arguments. For example, DIVIDE BYTE A B says that we are to take values referred to by A and B and divide them. Which is divided by what is determined by the specific Syntax of the DIVIDE BYTE instruction.

The subject and objects can include

1. The location where data to be processed is found.
2. The location where the result of processing is to be stored.
3. The location where the next instruction to be executed is found. (When this type of operand is not specified, the instructions are executed in sequence).

Instructions may be classified into categories such as input/output (I/O), data movement, arithmetic, logic, and transfer of control.

Input/output instructions are used to communicate between the unit and external devices.

Data movement instructions are used for copying data from one storage location to another and for rearranging and changing of data elements in some prescribed manner.

Arithmetic instructions permit addition, subtraction, multiplication, and division.

Logic instructions allow comparison between variables, or between variables and constants. Transfer of control instructions are of two types, conditional or unconditional.

Conditional transfer instructions are used to branch or change the sequence of program control, depending on the outcome of the comparison. If the out-come of a comparison is true, control is transferred to a specific statement number; if it proves false, processing continues sequentially through the program. Unconditional transfer instructions are used to change the sequence of program control to a specified program statement regardless of any condition.

What does MeccCode Look Like?

MeccCode is just a series of instructions, which are executed (acted on) one at a time. Each instruction tells the MotorVator to do one thing.

A MeccCode program looks like this:

```
Begin Program

Declare      Byte   Count = 1

Do

      Count = Count + 2
      DisplayHex Count

Until Count > 250

End Program
```


Instructions

Your program will be made up of a list of MeccCode Instructions. As each instruction is executed (or run), it will instruct the MotorVator to do something or check something. The main types of instructions are

Hardware Active Instructions.

These Include instructions to control the Motor Outputs, Servo and Action ports, update the displays and produce sounds from the speaker.

Hardware Input Instructions.

These include instructions to read the current state of all of the input ports (On/Off, Timed and Analogue).

Program Flow Instructions.

These instructions allow you to execute different parts of your program depending on the status of various inputs. These include a number of IF instructions, LOOP and WAIT.

Data Instructions.

These allow you to change the value of data variables within your program, so you can store data to be used later. These include instructions to Move, Add, Subtract, Multiply, Read and Write.

Event Instructions.

These are associated with establishing events. See Events Programming on page 69.

Bytes and Words

Within the MotorVator, values are stored for use in instructions. The MotorVator has only two different sized buckets into which values can be stored. These buckets are called Bytes and Words.

A Byte contains 8 single bits (each of which can be 1 or 0). Joined together, a Byte can contain from 00000000 to 11111111, or in decimal from 0 to 255. Because the Bitwise representation is cumbersome and hard to read, it is common to express byte value in Hexadecimal, where two hexadecimal "Digits" with value from 0..9, A..F can represent the value in one byte. Each Hexadecimal digit represents 4 bits (00 to 15 decimal, where "A" is 10, "B" is 11 etc). To distinguish Hexadecimal numbers from decimal numbers, add a 0x to the front, i.e. 53 is 53 decimal, 0x53 is 53 Hexadecimal (=83 decimal).

For example:

Bit	Hexadecimal	Decimal
00000000	00	00
00000101	05	05
00001100	0C	12
00010000	10	16
01011110	5E	94
11001101	CD	205

We can use a byte to store any value that will only require a range of between 0 and 255.

A Word is twice as big as a Byte, and has 16 bits. It can handle numbers in a range of 0 to 65535, so we use Words when the values might get bigger than 255. We also use a Word when we want the variable to point to another variable.

Some instructions expect a Byte, some expect a Word. If you try to use the wrong one, you'll get the wrong result!

Variables and Constants

Now that you know whether you want a Byte or a Word, you need to also know whether the instruction that you are using expects a Constant or a Variable.

A constant is just that - a fixed value that doesn't change. For example "Forward" is defined as a constant that is always used with the SET MOTOR command to indicate that you want the Forward direction. The Set Motor Command expects to see you referring to a Constant for the Direction. However, it does allow you to supply a Variable for the speed, which is good because you might want to use the results of a calculation as the speed.

Some instructions only read the values that you give them (i.e. the SET MOTOR command) and wont change them. Some instructions however are meant to change the values.

For example, the C = A + B instruction takes the value of Byte A, adds the value of Byte B and stuffs it into the location of Byte C. Now you can specify either Constants or Variables for A and B, but C HAS TO BE A VARIABLE.

Declaring Variables

In the Declare instruction, you can give an initial value that will be loaded into each variable when your compile program is downloaded to the MotorVator. Use this to define constants against which you can compare in `If Byte` and `If Word` statements. Note that there is nothing to stop you storing another value into the variable (either intentionally or accidentally) once your program is running.

The initial value is optional, but is only assigned the first time you download your code to the MotorVator. So if your program happens to write over then it won't be able to be relied upon in future executions. Make sure not to store data into your constants and overwrite values that you are relying on!

Type	Can hold	Example	Use for
Byte	A value from 0-255, or a single character	Declare Byte Count Declare Byte D_Chk "D"	Storing any value that won't exceed 255. Constants for comparisons (If s). The Optional Initial Value can be a decimal number, Hex Number (e.g. 0x6F) or single character in double quotes.
Word	A value from 0-65535	Declare Word WTime 1000	A variable used in any instruction that requires a word variable. A variable that will be used as a pointer to data.
Byte Array	A list of Byte values	Declare Byte Array Table 32,"B",0x3D, 35	Making a table of data. Use any valid byte constant values. The SIZE option creates an empty table
String	A String of Characters	Declare Tune as "A,B,C#6"	Play Tune strings. Data that needs to be entered as characters, not values. A Zero byte is always added to the end of a String variable.

Type	Can hold	Example	Use for
Constant	A name for a constant	Declare Constant SlewMotor 3	Give a logical name to a constant value. Here we think we are going to connect the Motor for the Slew action to Motor Output 3. Thus we can write SET MOTOR SlewMotor "F" Speed rather than SET MOTOR 3 "F" Speed. This is an advantage because if we need to change the physical connection to say motor 2, all we need to do is change the one Declare Constant SlewMotor 2 and all the places where we refer to this motor will be updated.

Use Declare Constant for Naming Constant values to go in Constant arguments, and Declare Byte Initial Value for Naming values to go in Variable Arguments.
Example:

```
Declare    Byte      SlewSpeed    = 40
Declare    Constant  SlewRight    = "B"
Declare    Constant  SlewMotor    = 1
           SetMotor  SlewMotor,   SlewRight,   SlewSpeed
```


Addresses and Pointers

When we want to store information, we have define a word or a byte area large enough to hold the information, and give it a name.

Think of this like a mailbox, with a number. We can put stuff into the box by simply saying "Put Letter into Mailbox 23", or take it out with a "Take a look into Mailbox 23 and tell me what's there".

Now think about a whole street full of mailboxes. If I wanted you to put a copy of the same newsletter into each one, I could say
"Put newsletter into Mailbox 1"
"Put newsletter into Mailbox 3"
"Put newsletter into Mailbox 5" and so on, right down the street.

But I'd probably prefer say something like
"go down this side of the street, and put a newsletter into each box, till you get to the end".

Now to do this we need to keep track of which mailboxes we've already been past.

To do this, we need to use the concept of a "Pointer". A pointer is just what is says - a piece of information that "points" to another piece.

So to use our newsletter example, we might say
"Start off at the first mailbox on this side of the street".
"Put a newsletter into the mailbox that you're pointing at"
"Now point to the next mailbox on this side"
"Put a newsletter into the mailbox that you're pointing to"
"Now point to the next mailbox on this side"
"Put a newsletter into the mailbox that you're pointing to"
"If we aren't at the end of the street, keep going"
etc

Or to be more concise

"Start off at the first mailbox on this side of the street".
Repeat:
"Put a newsletter into the mailbox that you're pointing at"
"Now point to the next mailbox on this side"
Until "at the end of the street"

We have instructions for managing pointers:

```
Newsletter_Pointer = SetDataPointer(Mailbox1)
```

is how we say "Start off at Mailbox1".

```
WriteData      Newsletter_pointer, Newsletter
```

is how we say "put a newsletter into the mailbox pointed to by Newsletter_pointer, then go on and point to the next mailbox"

Looking up a series of data is done with the ReadDataByte instruction, which says "take a look at the data in the mailbox pointed to by mailbox_pointer, copy it down to a notepad, then point to the next mailbox"

```
Notepad = ReadDataByte(Mailbox_pointer)
```


Program Flow

When writing your MeccCode program, keep in mind that the instructions will execute as fast as they can. For example, if you execute a SETSERVO Command, the MotorVator won't wait for the servo to go to the required position before executing the next instruction. As an example, this simple program looks like it will move Servo 1 to the left, then finish.

```
Begin program

Setservo      1,"B",75

End program
```

However, if you try it, you will see nothing appears to happen. This is because the END command resets all servos to the centre position. This is done so quickly after the SET Servo command that you don't see any movement. If you need to know that the command has completed before the next command is executed, use "Wait" or read in some sort of feedback from the analogue or digital ports and build a wait loop until the required position is reached, then proceed on, e.g.

```
Begin program
Declare byte endswitch

Setservo      1,"B",75
Do
       Endswitch = readonoff(1)
Until endswitch = closed


End program
```

Events Programming

The Events Construct is a very powerful way of programming real-time systems that need to act on inputs from external or other sources.
There are basically two ways in which we can write a real-time program that monitors a range of inputs and controls a range of outputs:

The "Main Loop" Method

Here we create a repeated "Loop" of instructions that read the inputs that we need to use, and then decide on what action to take.

Use the Main Loop to monitor inputs that change frequently. For example, you might have a program that reads the Director Joysticks and controls several motors on a crane.

```
Begin program

Loop

Luffspeed = readjoystick(1,"Y")
Luffdir = readjoystickdir(1,"Y")
If luffdir = "F" then
       Setmotor      1,"F",luffspeed
Else
       SetMotor      1,"B",luffspeed
End if

slewspeed = readjoystick(1,"X")
Slewdir = readjoystickdir(1,"X")
If slewdir = "F" then
       Setmotor      2,"F",slewspeed
Else
       SetMotor      2,"B",slewspeed
End if

End loop
```

The main loop reads the Joysticks as frequently as it can to ensure good responsiveness.

The "Event Method"

Under the event method, we establish "Events" that we want to execute each time there is a change in the input. Regardless of what our main program is doing, when there is a change in the input, our Event Code will "Interrupt" the main program and allow us to execute our special event code.

Events are good for tracking inputs that change infrequently. A good example is a safety switch that would normally be open, but when closed indicates that something needs to be shut down immediately. You could put a test for the switch status in the main loop, but this would (i) add an overhead to the main loop that almost always would be unnecessary, (ii) mean that you would only detect the safety switch at one point in the main loop and you may need to act more quickly than this would allow. So we code an Event to trigger from any change in the safety switch input.

In general you use a combination of the "Main Loop" to monitor the frequently required inputs, and Events to handle the infrequent but important inputs.

Example: monitor the Director Joysticks and operate our crane, but if the switch on the "Rope Slack" ever closes, then we want to stop immediately to avoid the main rope tangling.

```
Declare eventhandler ropeslack

Declare byte ropestate
Ropestate = readonoff(1)
If ropestate = 1 then
       EmergencyStop
End if
End eventhandler
;-----------------------------------

begin program

setevent      onoff1event, ropeslack
enable events

Loop

drumspeed = readjoystick(1,"Y")
drumdir = readjoystickdir(1,"Y")
If luffdir = "F" then
       Setmotor      1,"F",drumspeed
Else
       SetMotor      1,"B",drumspeed
End if

slewspeed = readjoystick(1,"X")
Slewdir = readjoystickdir(1,"X")
If slewdir = "F" then
       Setmotor      2,"F",slewspeed
Else
       SetMotor      2,"B",slewspeed
End if

End loop

End program
```


Writing and Compiling Your Programs

MeccCompiler III

Meccanisms MeccCompiler is a traditional text-based compiler.

Use your favourite text editor to write your program code. You can use MeccCompiler's Source Window if you don't have another editor. The Source Document (source is the name for your code before it is "compiled" – processed – into a MotorVator-Readible numeric form) should be simple text, with no formatting commands.

Each instruction must start on a new line. You may leave blank lines between instructions.

If you want to put in a comment in plain english, start it with a Semicolon (;). Any text from the Semicolon to the end of the line will be ignored by the Compiler.

Introduction to MeccCode Programming

Introduction

This manual is designed to help you become familiar with the programming the Motorvator. It is designed to progressively introduce various concepts of programming in a logical sequence that builds on previous steps, so if you are new to programming We suggest you work through this in order. Experienced programmers will find this useful to skim through, possibly loading the complete sample programs provided in the disk as a start for further experimentation.

We will work with a real program rather than abstract fragments of some larger application, as this helps you gain experience with the whole cycle of writing, compiling, debugging and running your own applications.

Electronic Dice

Install the MotorVator Software from the CD. Run the MeccCompiler (Program>Meccanisms>MeccCompiler II). Select "New Source File", and choose your source file as the input source file.

Lets start off with a very simple program that turns the Motorvator into an electronic dice. Push a button and the LED digits will count rapidly, push another button and it will stop and display a number between 0 and 99.

Step One: Getting a number to display

Type the following text into the editor

```
Begin program

DisplayNumber 45

End program
```

What this will do seems fairly obvious, so Compile it. (Compile>Compile Source), then download it (MotorVator>Open Comms Window, MotorVator>Download Current Program) to the Motorvator.

Now Press the Green RUN radio button in the Comms Window (or press the Green ON MotorVator button twice).

When you run the program the Motorvator does it's four beeps, shows 99 – (the program number) and…. Goes straight to displaying 'on'. What went wrong?

In fact nothing went wrong. The Motorvator displayed the number, then immediately after, finished running your program and returned to the ON prompt.

Lets add a 'wait' instruction to force the Motorvator to slow down to human time. Your code should now look like this:


```
Begin program

DisplayNumber 45
Wait 500
End program
```

Try it.

Great! We can see the number 45 displayed for 5 seconds before the program ends.

The Wait For Time 500 doesn't read like a 5 second wait. So lets take some time to make sure our code can be understood better. Even professional programmers can't figure out what their code does months or years later so rely on carefully chosen variable names and comments to make the program more readable.

First let's add a comment to the 'Wait For Time' line thus:

```
Begin program

DisplayNumber 45
Wait 500            ; wait for 5 seconds
End program
```

Notice the semi Colon ';' at the beginning of the 'Five Second' comment. This tells the compiler that everything afterwards is for humans only and it should not try to interpret it as part of the instruction.

We can still make this more readable, (And more useful later on) by introducing a variable:

```
Begin program
Declare word delay  = 500

DisplayNumber 45
Wait delay          ; wait for 5 seconds
End program
```


Let's examine this change carefully. We have a new statement 'Declare Word', which is known as a Directive. In this case it tells the compiler we want it to use an algebraic word called Delay and it should be set the value of five hundred. We have then used it to replace the 500 in the instruction 'Wait For Time'. This makes the code much easier to read – especially later when the program is more complex. Variables have other more powerful uses as you will soon see.

Step Two: Getting the Number to Change

OK so we can display a number. A good start, but not very useful for the Electronic Dice program if the answer is always the same, so lets focus on getting the number to change.

First we need to change our program so the display number statement can display numbers other than 45. We start to do this by introducing another variable thus:

```
Begin program
Declare word delay  = 500
Declare byte dicevalue

DisplayNumber dicevalue
Wait delay          ; wait for 5 seconds
End program
```

If you try running this code, you will note the 45 has gone and been replaced by a 00. This is because we did not tell the compiler what value we wanted DiceValue to be, so it assumed a value of zero.

> It is good programming practice never to assume what the value of a variable is, and declare it to be a specific value at the start. Otherwise you can inadvertently introduce difficult to find bugs that seem to appear and disappear randomly.

Lets introduce some math to make our number change:

```
Begin program
Declare word delay  = 200
Declare byte dicevalue

DiceValue = DiceValue + 1
DisplayNumber DiceValue
Wait delay          ; wait for 2 seconds
End program
```

Now we need to set up a 'loop' so the program increments DiceValue , displays the number and loops around to do it all over again.
Our changed code looks like this:

```
Begin program
Declare word delay  = 100
Declare byte dicevalue

Loop
      DiceValue = DiceValue + 1
      DisplayNumber DiceValue
      Wait delay          ; wait for 2 seconds
End Loop
End program
```

The instruction "Loop" (and the matching "End Loop") tells the Motorvator to keep looping (repeating) all instructions between, forever.

> It is good programming practice to indent all of your code within a loop (or IF other conditional structure). It makes no difference to the compiler, but sure is easier to read for humans.

Run this code and you now have a slowly incrementing count on the display. Each second the digit is incremented. But it never ends! Once it gets to 99 it goes back to zero, Gets to 99 a second time, zero, THEN once it gets to 55 – zero again – Why?

A byte variable can contain a maximum value of 255 before an increment takes it back around to zero. (A useful feature in some programs). The display however, can only show two digits, thus

numbers between 100 and 200 show as 1 to 99, but because of the max value of 255, it cycles around to zero the third time through.

Lets add code to use a button to stop the count:

```
Begin Program
Declare Word   delay   = 25
Declare Byte DiceValue
Declare Byte Button

Do
        DiceValue = DiceValue + 1
        DisplayNumber DiceValue
        Wait delay

        Button = ReadButton(UpButton)

Until Button <> 0

End Program
```

As you can see we have added a third variable declaration Button, added a 'ReadButton' instruction and replaced the loop with an Do..Until Statement.

The ReadButton statement fetches the value of the left hand Up Bbutton on the MotorVator and places it in variable Button. The Until Statement checks to see if Button is not equal to Zero (No Press) and jumps back to the Do, if so.

In other words, if the button is not pressed, keep looping until it is.

We have some final adjustments to our code and it is complete:

(i) Change the Delay declaration to 1, so we count faster,
(ii) and add a delay so we can see the number
(iii) Convert the number to a range of 1...6, so we can use it as a dice value. We do this with the X= y MOD z command, which divides Y by Z and places the remainer in X.


```
Begin Program
Declare Word  delay  = 1
Declare Byte DiceValue
Declare Byte DiceDisplay
Declare Byte Button

Loop
       Do
       DiceValue = DiceValue + 1
       DiceDisplay = DiceValue Mod 6
       Increment DiceDisplay
       DisplayNumber DiceDisplay
       Wait delay          ; wait for 2 seconds

              Button = ReadButton(DownButton)

       Until Button <> 0

       Wait 500

End Loop

End Program
```

Now you have mastered the important basics. Try some of the following changes yourself. Experimenting like this is how even experienced professional programmers gain an understanding of unfamiliar commands or techniques.

1. Change the read button to other buttons
2. Change the count direction to down using the Subtract command
3. Get the program to wait for a button press before starting, and another to stop.
4. Add an additional delay and loop after the final count is displayed to start the program again automatically
5. Introduce a noise when you press the button with Play sound
6. Try introducing deliberate mistakes and see what the compiler/Motorvator does.

Debugging Your Program

Errors caused by faulty logic and coding mistakes are referred to as "bugs." Finding and correcting these mistakes and errors that prevent the program from running and producing correct output is called "debugging." Rarely do complex programs run to completion on the first attempt.

Often, time spent debugging and testing equals or exceeds the time spent in program coding. This is particularly true if insufficient time was spent on problem definition and logic development.

Some common mistakes which cause program bugs are: mistakes in coding punctuation, incorrect operation codes, transposed characters, keying errors and failure to provide a sequence of instructions (a program path) needed to process certain conditions. To reduce the number of errors, you will want to carefully check your program design before getting down to actually keying it in.

After the program has been checked visually accuracy, the program is ready to be compiled. The compiler prepares your program (source program) to be executed by the Motorvator and it has certain error diagnostic features which detect certain types of mistakes in your program. These mistakes must be corrected. Even when an error-free pass of the program through the compiler program is accomplished, this does not mean your program is perfected. However, it usually means the program is ready for testing in a MotorVator – perhaps using the built in debugger that allows you to step each line of code and check that the results are what you intended before continuing with the next step.

The CLI Debugger

The CLI debugger allows you to debug your program by "Single stepping". This means that you can ask the Motorvator to execute one MeccCode instruction, then wait. While waiting, you can check the outcome of the last instruction, look at the variables in memory, and even change the values.

As an example, let's look at a sample program as follows:

```
Begin Program
Declare Word                              delay
Declare Byte DiceValue
Declare Byte DiceDisplay
Declare Byte Button

Loop
  Do
    DiceValue = DiceValue + 1
    DiceDisplay = DiceValue Mod 6
    Increment DiceDisplay
    DisplayNumber DiceDisplay
    Wait delay                                                    ; wai
    Button = ReadButton(DownButton)

  Until Button <> 0

Wait 500

End Loop

End Program
```

When compiled, this gives us an object map as follows:

```
[0000]/0000 Begin Program:  NOP      ;Begin Program
[0001]/0001                 CLEAR WORD                    %SysTemp      ;Defaulting %SysTemp
[0004]/0004                 CLEAR WORD                    %SysTemp2     ;Defaulting %SysTemp2
[0007]/0007                 MOVE WORD   :Loop             W_1, Program_delay
[000C]/0012 Line7:          NOP
[000D]/0013 Line8:          NOP
                                        ;Do
                                        Program_DiceValue, B_1, Program_DiceValue
[000E]/0014                 ADD BYTE    ;DiceValue = DiceValue + 1
                                        Program_DiceValue, %SysTemp
[0015]/0021                 MOVE BYTE   ;Prep for Mod Byte   B_6, %SysTemp2
[001A]/0026                 MOVE BYTE     ;Prep for Mod Byte  %SysTemp, %SysTemp2
[001F]/0031                 DIVIDE BYTE ;DiceDisplay = DiceValue Mod 6
                                        %SysTemp2, Program_DiceDisplay
[0024]/0036                 MOVE BYTE   ;Cleanup after Mod Byte  Program_DiceDisplay
[0029]/0041                 INCREMENT BYTE  ;Increment DiceDisplay
                                        Program_DiceDisplay
[002C]/0044                 DISPLAY NUMBER  ;DisplayNumber DiceDisplay
[002F]/0047                 WAIT FOR TIME   ;Wait delay     Program_delay
[0032]/0050                 READ BUTTON     B_DOwnButton, Program_Button
                              ;Button = ReadButton(DOwnButton)
[0036]/0054                 IF BYTE EQUAL   Program_Button, B_0, Line8
                              ;Until Button <> 0
[003D]/0061                 WAIT FOR TIME   ;Wait 500       W_500
[0040]/0064                 JUMP            Line7
                                        ;
[0043]/0067 End Program:    END         ;End Program
[0044]/0069 #SysTemp:                   0,0                 ;Used Variable
```


This shows the AWOS Instructions that the MotorVator will execute, and all the program space addresses at which the code and data reside.

Now we don't expect you to want or be able to decipher every line in this object listing. It is however useful to be able work out the storage location for variables, so that you can check of what your program is doing.

You can "Single Step" the program on the MotorVator, which means you can have the MotorVator execute one instruction, then stop till you tell it to execute the next instruction.

To do this, after downloading your program, stay in the MotorVator Communications window, and enter "D" at the Command Prompt.

This will start your program running from the first instruction, and display that instruction.

Pressing ENTER will cause one more instruction to be executed.

At any time you can double click on the addresses shown in the debug window to get MeccCompiler to show you which instruction you are looking at.

MeccCode III Reference

New Instructions in MeccCode Release III (December 2006)

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 07300300000000000000000
; 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.

Declare

See Also: Variable and Function Names, When to Declare, Number Formats

Declare Byte

Declare Byte *?Name?* [= *?Value?*]

Declares a byte variable. Can be used at the top of the program to declare a global variable, (one available to all code), or within a function declaration to create a local variable only available to code within that specific function.

?Name? represents the name of the variable you wish to declare. It cannot be a reserved word, (Eg Declare), cannot contain spaces and must be unique within the same scope. That is no two globals may have the same name and no two local variables can be identical. A local variable CAN have the same name as a Global, and as such will be used in preference to the Global for all code within that function. A warning will be issued by the compiler in these circumstances as if unintentional – this could be a difficult error to trap.

?Value? represents the optional value you wish to assign to the variable every time this program is run or this procedure is entered. It can be either a literal (Number 0-255), a single String character, (e.g. “F”), or the Name of an acceptable constant. Other variable names or equations are not allowed.

If ?Value? is omitted, variable is not explicitly set when program runs. This generates a warning at compile time. Not setting a default can be useful if the variable is a global and it is desirable to retain the value from one run to the next. Locals are created dynamically as required and their value cannot be guaranteed at all.

Examples:

```
Declare Byte Mybyte          ;declares a byte called mybyte

Declare Byte DefaultedByte = 0    ;declares and defaults a byte
variable
```

Declare Word

Declare Word *?Name?* [= *?Value?*]

Declares a word variable. Can be used at the top of the program to declare a global variable, (one available to all code), or within a function declaration to create a local variable only available to code within that specific function.

?Name? represents the name of the variable you wish to declare. It cannot be a reserved word, (Eg Declare), cannot contain spaces and must be unique within the same scope. That is no two globals may have the same name and no two local variables can be identical. A local variable CAN have the same name as a Global, and as such will be used in preference to the Global for all code within that function. A warning will be issued by the compiler in these circumstances as if unintentional – this could be a difficult error to trap.

?Value? represents the optional value you wish to assign to the variable every time this program is run or this procedure is entered. It can be either a literal (Number 0-65355), two string characters, (“AB”) or the Name of an acceptable constant. Other variable names or equations are not allowed.

If ?Value? is omitted, variable is not explicitly set when program runs. This generates a warning at compile time. Not setting a default can be useful if the variable is a global and it is desirable to retain the value from one run to the next. Locals are created dynamically as required and their value cannot be guaranteed at all.

Examples:

```
Declare Word MyWord              ;declares a word called
myword

Declare Word DefaultedWord = 0   ;declares and defaults a Word
variable
```

Declare Boolean

Declare Boolean *?NAME?* [= *?Value?*]

Declares a Boolean variable. Boolean is a special type of Byte variable that can only interpret its value as True or False. True and False are system declared constants. False is defined as 0 and True is any other byte value. Boolean Variables can be used in place of an equation for Until, While and If clauses. Therefore a function defined as a Boolean can provide a powerful way to make decisions in your code.

Declare Boolean can appear at the top of the program to declare a global variable, (one available to all code), or within a function declaration to create a local variable only available to code within that specific function.

?Name? represents the name of the variable you wish to declare. It cannot be a reserved word, (Eg Declare), cannot contain spaces and must be unique within the same scope. That is no two globals may have the same name and no two local variables can be identical. A local variable can have the same name as a Global, and as such will be used in preference to the Global for all code within that function. A warning will be issued by the compiler in these circumstances as if unintentional – this could be a difficult error to trap.

?Value? represents the optional value you wish to assign to the variable every time this program is run or this

procedure is entered. It can be either True or False. Other variable names or equations are not allowed.

If ?Value? is omitted, variable is not explicitly set when program runs. This generates a warning at compile time. Not setting a default can be useful if the variable is a global and it is desirable to retain the value from one run to the next. Locals are created dynamically as required and their value cannot be guaranteed at all.

Examples:

```
Declare Boolean MyFlag                  ;declares a boolean variable called myFlag

Declare Boolean IsRunning = True        ;declares and defaults a boolean variable to system constant
```

Declare Constant

Declare Constant *?NAME? = ?Value?*

Common convention is constants' names should be all uppercase to differentiate from variables. The value is not optional in declaration. Constants will throw a compiler exception if any code attempts to change them.

?Name cannot be a reserved word, (Eg Declare), cannot contain spaces and must be unique within the same scope. That is no two globals may have the same name and no two local constants can be identical. A local constant CAN have the same name as a Global, and as such will be used in preference to the Global for all code within that function. A warning will be issued by the compiler in these circumstances as if unintentional – this could be a difficult error to trap.

System predefined constants are considered global and cannot be overridden by re-declaring them as some other value – either at global OR local level.

?Value? Can be any number or string literal value appropriate for its eventual use in functions, statements and/or variable assignments.

Constants are very useful for setting overall boundary values. Then if this value needs to be changed later it only needs to be changed in one place.

Constants can also be used in Data Tables

Example:

```
Declare Constant MAX_ANGLE = 34     ;declares and sets a constant
```

Declare DataTable

Declare DataTable *?Name?* [[= *?Value?[,?value?]…*] /OR/ [Reserve ?NumberOfBytes?] .....[= *?Value?*] /OR/ [Reserve ?NumberOfBytes?]

End DataTable

Data table declarations are generally used to describe a large quantity of data and as such can span multiple lines. A Carriage return is treated as a comma so the whole block is seen by the compiler as one contiguous statement. They are commonly used in conjunction with Data Pointers, ReadData and WriteData commands.

Data Tables are always global and cannot be declared inside function or event declarations as memory space restrictions make local Data Tables unfeasible.

?Name cannot be a reserved word, (Eg Declare), cannot contain spaces and must be unique within the same scope.

?Value? can be substituted with literal values or constants including strings but not Variable Names. The **Reserve**

keyword can be used to set aside a specific number of bytes.

Example:

```
Declare DataTable MyTable               ;MyTable -  lots of data
                  25,26,27              ;Three Byte values
                  Reserve 1024          ;Reserve 1024 bytes
                  TRUE, MY_CONSTANT     ;2 constants in table
                  "My String Goes Here" ;A String in the Table
End DataTable
```

Declare DataPointer

Declare DataPointer *?Name?*

Declares a special type of variable. This variable contains no useful data directly, but is used to point to a memory address where the desired data resides. It is used to work with DataTables.

Data Pointers can be declared globally or locally.

?Name? represents the name of the pointer you wish to declare. Data Pointer names cannot contain spaces. It cannot be a reserved word, (Eg Declare) and must be unique within the same scope. That is no two globals may have the same name and no two local data pointers can be identical. A local pointer CAN have the same name as a Global, and as such will be used in preference to the Global for all code within that function. A warning will be issued by the compiler in these circumstances as if this naming was unintentional it would make a difficult error to trap.

Data Pointers can only be assigned using the SetPointer(?VariableName?) command. ?VariableName? can only be the name of a valid, explicitly declared variable of type DataTable. A Pointer cannot be used to point to functions or other code, However wayward use of WriteData or ReadData could move a pointer past the end of a variable and over other variable declarations or even code. Pointers should be used with extreme care!

Examples:

```
Declare DataPointer MyDP                ;declares a data pointer variable called myDP

MyDP = SetPointer(MyTable)              ;Example of use of the setPointer command to assign the pointer
```

Declare Function

Declare Function *?Name?*([[*?Arg1?* as $Type$]….[*?Argn?* as $Type$]]) [Returns [Byte /OR/ Word /OR/ Boolean /OR/ DataPointer]]

$Statements that make up the function body$

[Return ?Value?]

[Return ?Another Value?]

End Function

Declares a user defined function that can be called by the main program or any other function within the program. Functions must be declared before or after then end of the main program and not within other function, event or data table declarations.

?Name? represents the name you want to use for this function declaration. It cannot be a reserved word (eg

Declare) and must be unique. Function names cannot contain spaces.

Function declarations are by nature strictly global declarations and cannot be nested. That is one function declaration cannot contain another function declaration.

Functions declared that take no arguments and return no variable must still be declared with brackets after the name. There must be no spaces after function name and before the brakets eg:

```
Declare Function MyFunction()
   ;Your code here
End Function
```

$Statements that make up the function body can be any allowable declaration, statement, system function or call to other functions defined within this program – including those within files imported into this program using the Include File statement.

?arg1?, ?argn? represent variables the function expects to be passed when it is called. A function may require any number of arguments to be passed when it is called including none. Arguments are always passed by value – that is the function is called with a copy of each argument. If the function alters one of these values it does not alter the value back in the calling routine. This includes Globals passed to the function. To alter a Global within a function - reference it directly

There are only two ways a function can provide altered values to its calling routine:

1. The value the function returns (if declared)
2. Directly altering a globally declared variable within the function.
3. Clever manipulation of the user stack – for serious advanced users only

$Type$ the declared arguments to the function can be of type Byte, Word, Boolean or DataPointer. Functions, DataTables cannot be passed to a function and declaring an argument as a constant does not make sense. Constants and Literal values can be passed to a function as substitutes for any arguments however.

Functions that are declared as returning a Boolean data type can be used as the evaluation for If, While and Until control structures.

Returns – declares if the function returns a value or not. If omitted calling this function using the format MyVal = MyFunction(….) will generate an error. Functions that do not return a value must be called using the Call statement as follows:

```
Call MyFunction(…..)
```

Please note using Call with a function that does return a value will generate a compiler error.

Functions that are declared as returning a value can use the Return keyword to return any valid variable, constant or literal that matches the declared return type as ?Value?. Functions declared as returning a value, that do not explicitly use the returns keyword, (or the keyword is perhaps inside an If statement not accessible on this run), will automatically return a zero to the calling routine when the End Function statement is encountered.

An Example Function Declaration:

```
Declare Function DoMath(var1 as Byte , var2 as Byte, MathType as Byte) Returns Word

    Declare Word RtnVal = 0          ;Declare a local word variable to contain a calculated return value
    Declare Byte Temp = 0

    If MathType = 1 Then             ;1 means Multiply
```



```
        RtnVal = var1 * var2
    Else
        If MathType = 2 Then        ;2 means Divide
            Temp = var1 / var2
            RtnVal = MakeWord (0,Temp)  ;Convert result to a word
        Else
            Return O
        End If
    End If

    Return RtnVal

End Function
```

This function can be called as follows:

```
Myresult = DoMath(myvar, 12, 2)        ;divide variable byte 12
```

Whereas using the call statement with a function that returns a variable will generate an error:

```
Call DoMath(myvar,anothervar,1)     ;Ignores return variable and generates an error
```

The return statement is used to return variables from a function (RtnVal in the example above). But consider this:

```
……

Return      ;No variable supplied so will return the default 0

End Function
```

The return statement here does not return a value and is therefore not necessary as the End Function will return the default value automatically. In this case the compiler will remove the return statement as part of its optimisation. You

should only use returns in your code if you want to exit early or return a specific variable

Declare EventHandler

Declare EventHandler *?EventHandlerName?***()**

$Statements that make up the EventHandler body$

[Return]

End EventHandler

Declares a user defined function of a special type that can only be called by the the operating systems events handling services. Event Handlers must be declared before or after the end of the main program. Event Handlers cannot be declared inside other event handlers, Function or Data Table declarations.

?Name? represents the name you want to use for this EventHandler declaration. It cannot be a reserved word (eg Declare) and must be unique. EventHandler names cannot contain spaces, take arguments or return values

EventHandler declarations are by nature strictly global declarations and cannot be nested. That is one EventHandler declaration cannot contain another.

$Statements that make up the EventHandler body$ can be any allowable declaration, statement, system function or call to other functions defined within this program – including those within files imported into this program using the Include File statement.

You *can* have multiple events going to one handler – but there is no way to determine which event caused the handler to be invoked. Recommended practice is to have a

dedicated event handler for each event you wish to work with.

Caution calling code from within event handler. The compiler will ensure all local variables are safe, but if your event code alters a global variable that was in the process of being used by the normal program you may end up with a bug that is very difficult to find.

Consider the following code fragment:

```
Declare Byte MyGlobalVar = 99
Declare Byte AnotherVar = 0

Begin Program

SetEvent EvMicroswitchHandler, OnOff1Event

   While MyGlobalVar <> 1

         AnotherVar = MyGlobalVar + 1

          << Busy doing something with the local variable here >>

         MyGlobalVar = AnotherVar

   End While

   << More really cool code here >>

End Program

Declare EventHandler EvMicroswithHandler()

   <<Do something useful here>>

   MyGlobalVar = 1                ;Signal something has happened

End EventHandler
```


In the somewhat contrived example - if you imagine the intent of the code was to loop around doing something - including modifying MyGlobalVar - until a microswitch attached to port OnOff1 was pressed when it would exit the While /End While and go do something else.

If the microswitch was closed (thus triggering the event), while the program was working hard in the section noted << Busy doing something with the local variable here >> then the following would happen:

1. Event handler would be called, doing something and finally setting MyGlobalVar to 1
2. Code in the main program would resume execution and eventually get to the bit that assigns MyGlobalVar to AnotherVar
3. The While / End While would never exit because it would not be set to 1!

TIP: Any global variables used by event handlers should only be altered by the event handler, or used as simple set, reset flags

Variable and Function Names

All variable, constant and Function names must follow the following rules:

1. Must start with an alpha character (a-z)
2. Cannot contain any of the recognised math or relational operators such as +-*/<>=
3. Cannot be a reserved word, (A System function or statement name).
4. Cannot contain Brackets (or) or comma's or quotes.

5. Must be more than 2 characters long
6. Other symbols and numerals can form part of the name, but not the first character. For Example: MyFunction1 or Good1_Name6

See Also: Declarations, When to Declare, Number Formats

When To Declare

All variables and constants must be declared within the program. That is the file being compiled or any files included using the Include File statement.

Because the compiler checks right through the whole program for variable declarations before checking each line for errors, variables can be declared before or after use.

However it is good programming practice to declare variables before you use them - ie global variables at the top of the program before the Begin Program statement and local variables at the top of the routine they will be used in.

For more information see Program Layout

Number Formats

Wherever it is legal to place a number it is also possible to place a hexidecimal number by prefixing the number with a zero and suffixing with an H.

Example:

```
MyByte = 0FFh          ;Equivalent of decimal 255
MyByte = 12 + 0Fh      ;Not recommended to mix formats in same statement, but quite legal
```


System Functions

These functions are provided by the operating system and do not need to be defined in your code to be used. Functions follow the same general format as a user defined function that returns a value.

```
VarName = SystemFunctionName(arg1, arg2, .....argN)
```

See Also: Statements

Compliment

?MyByte? = **Compliment (***?ByteToCompliment?***)**

Returns a bitwise ones compliment of ?ByteToCompliment?. Generally useful only in specialist bitwise operations or as an alternative way to flip a true flag false and visa versa.

See Also: Not()

GetClock

?MyWord? = **GetClock()**

Returns the current value of an internal timer running at 1024hz. This is a Word value that rolls through 0 after 65535 and runs continuously. This is useful where it is important to your code to know how long between doing things. For example - Set a motor going, GetClock and note its value, wait for a limit switch to close and get the clock again to see how long it took to get there for further analysis.

GetProgramNumber

?ByteVarName? = **GetProgramNumber()**

Reads the program number that was selected before pressing the Green RUN key on the MotorVator.

There are no arguments.

Selecting any Program number from 90 – 99 will invoke the current user program so this instruction is useful to allow your program to perform different behaviour depending on how it is invoked. For example program 99 could be used to run the program while some of the other numbers are used for debug modes, or variations on the operational mode as decided by your code.

Example:

```
MyProg = GetProgramNumber()

If MyProg = 98 then

   Call   Run_Diagnostics()      ; if you select 98, then you can run your diagnostic code first

endif

'  main program here             ; then your main program can run

.....
```

HighByte

?ByteVarName? = **HighByte(***?WordVarName?***)**

Returns the value of the upper of the two bytes that make up a word. For example:

```
WordVar = 259

MyByte = HighByte(WordVar)
```

Where MyByte will be set to 1 (And the lower byte will be zeros)

Useful for determining a byte overflow situation after byte multiplication.

See Also: LowByte()

LowByte

?ByteVarName? = **LowByte(***?WordVarName?***)**

Returns the value of the lower of the two bytes that make up a word. For example:

```
WordVar = 259

MyByte = LowByte (WordVar)
```

Where MyByte will be set to 3 (And the high byte will be 1)

Useful for retrieving a byte value after byte multiplication where the result is unlikely to go over 255, or you don't care about a value higher than 255.

See Also: HighByte()

MakeWord

?WordVarName? = **MakeWord(***?ByteVarName?***)**

Converts the byte specified into a word variable.

See Also: HighByte, LowByte

Not

?MyByte? = **Not(***?ByteValue?* /OR/ *?WordValue?***)**

Returns a True or False based on the contents of the supplied variable. For example: If the supplied variable is a non zero value, the function returns False, and visa versa.

NOTE: This is the only system function that can be used with a block structure statement like IF as follows:

```
If  Not(MyByte) Then

        ......

End If
```

Pop

?ByteVarName? /OR/ *?WordVarName?* = **Pop()**

Retrieves a byte or word, as appropriate to the supplied variable name, from the user stack. Pop, used in conjunction with Push, allows your program to store data temporarily.

NO error will be generated if you push a series of bytes and pop a word, but it may be a difficult bug to track.

The stack is only used for data and no program addresses, (such as return addresses), are placed there.

The stack *IS* used by the compiler to store values before, during and after a user defined function call, so you should use these commands with great care.

See Also: Push

ReadAnalogue

?ByteVarName? = **ReadAnalogue(***?AnaloguePort?***)**

Reads the current state of the Analogue port specified by ?AnaloguePort?. This should be a literal number or constant with a value between 1 and 6, any other value will generate a compiler error. Variable names are not permitted.

For example:

```
MyByte = ReadAnalogue(1)     ;returns the value of Analogue port 1.
```

See Also: ReadJoystick

ReadBattery

?ByteVarName? = **ReadBattery()**

Reads the current value of the internal Battery as a byte number between the value of 0 and 255 where 255 equals a full 9 volt battery.

Please note if no battery is present then unreliable values may be returned.

ReadButton

?ByteVarName? = **ReadButton(***?ButtonConstant?***)**

Returns the current value of the button specified by ?ButtonConstant?. This value must be a literal number from 1-4. It is recommended you use the following system constants to make your code more readable. (NB: These are listed in numerical value order)

Joystick1Button1 (on Director)

Joystick1Button2

Joystick2Button1

Joystick2Button2

See Also: SetEvent - to read MotorVator buttons

ReadData

?VarName? = **ReadData(***?DataPointerName?***)**

Reads data from the position pointed to by DataPointerName, and the pointer is incremented by one for a Byte or Boolean read or two when used to read a word or datapointer from the table.

?DataPointerName? should have been previously initialised with the ?DataPointerName? = SetDataPointer(?DataTable?)

?VarName? can be a Byte ,Word , Boolean or DataPointer variable.

Note using this function with a datapointer that is not pointing into a valid data table will result in unpredictable results.

See Also: WriteData

ReadJoystick

?ByteVarName? = **ReadJoystick(***?JoystickNumber?,?XorY?***)**

Retrieves the current value of the joystick X or Y axis, as specified by ?JoystickNumber? and ?XorY?. These should be a literal number or constant with a values of 1 or 2, “X” or “Y” respectively, any other values will generate a compiler error. Variable names are not permitted.

Note that ReadJoystick returns a 'normalised' reading that represents the percentage of movement away from the centre position. Use ReadJoystickDir to determine in which direction the stick has moved.

Also, see Calibrate Joystick.

For example:

```
MyByte = ReadJoystick(1,"X")     ;gets the value of Joystick 1, X axis
```

See Also: ReadJoystickDir, Calibrate Joystick

ReadJoystickDir

?ByteVarName? = **ReadJoystickDir XE "Joysticks:Direction" (***?JoystickNumber?,?XorY?***)**

Retrieves the current direction of the joystick X or Y axis, as specified by ?JoystickNumber? and ?XorY?. These should be a literal number or constant with a values of 1 or 2, “X” or “Y” respectively, any other values will generate a compiler error. Variable names are not permitted.

The returned direction value will be a “F” for forward or “B” for backward

For example:

```
MyByte = ReadJoystickDir(1,"X")     ;gets the direction of Joystick 1, X axis
```

See Also: ReadJoystick

ReadOnOff

?ByteVarName? = **ReadOnOff(***?OnOffPort?***)**

Reads the current state of the OnOff port specified by ?OnOffPort?. This should be a literal number or constant with a value between 1 and 4, any other value will generate a compiler error. Variable names are not permitted.

The result will be either a 0 (for Open) or 1 (for Closed)

For example:

```
MyByte = ReadOnOff(1)     ;returns the value of OnOff port 1.
```

See Also: ReadTimedState

ReadPulseCountDigital

?ByteVarName? = **ReadPulseCountDigital(***?OnOffPortNo?***)**

Returns the number of full pulse cycles received on the specified port since last read. This is useful for counting low frequency pulses such as those received from counting micro-switches. The pulses on these ports are sampled approximately 110 times per second, so any frequency

higher than 60hz is unlikely to be reliably tracked. Note also the value returned is a byte variable which will cycle round after reaching 255.

See Also: ReadPulseCountTimed

ReadPulseCountTimed

?ByteVarName? = **ReadPulseCountDigital(***?TimedPortNo?***)**

Returns the number of full pulse cycles received on the specified port since last read. This is useful for counting low frequency pulses such as those received from counting micro-switches. The pulses on these ports are sampled approximately 110 times per second, so any frequency higher than 60hz is unlikely to be reliably tracked. Note also the value returned is a byte variable which will cycle round after reaching 255.

See Also: ReadPulseCountDigital

ReadTimedPulseWidth

?ByteVarName? = **ReadTimedPulseWidth(***?TimedPort?***)**

Reads the value of the last pulse width received on the port specified by ?TimedPort?. This should be a literal number or constant with a value between 1 and 4, any other value will generate a compiler error. Variable names are not permitted.

Value returned is a linear scalar number that approximates the number of 110hz pulses counted while the timed port was high for the last full pulse. That is – if the port is in the middle of receiving a pulse, you will still get the count for the last fully received pulse.

Note also that the maximum pulse width that can be measured is approximately 2.5 seconds (255/110)

For example:

```
MyByte = ReadTimedPulseWidth(1)     ;returns the PW value of Timed port 1.
```

ReadTimedState

?ByteVarName? = **ReadTimedState XE "Ports:Timed:Reading State" (***?TimedPort?***)**

Reads the current state of the Timed port specified by ?TimedPort?. This should be a literal number or constant with a value between 1 and 4, any other value will generate a compiler error. Variable names are not permitted.

The result will be either 0 (Port Open) or 1 (Port Closed)

For example:

```
MyByte = ReadTimedState(1)     ;returns the value of Timed port 1.
```

See Also: ReadOnOff

Receive

?mmmName? = **Receive(***?CommPort?***)**

For future implementation.

See Also: Transmit

Randomize

?MyByte? = **Randomize()**

Provides a pseudo random seed number between 1 – 255. This number is generated by combining several internal system variables including the state of the interrupts. Note: If you continually call this function with little other code in operation the responses will not appear as random as when the function is used within the context of a larger program that calls it occasionally.

SetPointer

?DataPointerName? = **SetPointer(***?DataTableName?***)**

SetPointer can only be used to set the address the specified datapointer points to.

Please see notes on declaring data pointers for more information.

See Also: Declare DataTable,

Until

Until *?ValidBooleanExpression?*

See Do / Until in Looping and Conditional Structures Section

See Also: Condition Syntax

While

While *?ValidBooleanExpression?*

See While / End While in Looping and Conditional Structures Section

See Also: Condition Syntax

Statements

Statements differ from system functions in one important way - they do no return any value. Therefore they do not require an associated variable name or brackets. The format is:

```
Statement arg1, arg2, ......argN
```

See Also: System Functions

Beep

Beep ?Note?, ?Duration?

Note is a relative tone - Smaller numbers are higher pitched.

Duration is in 1/10 of a second.

See Also: PlayTune

Begin Program

Marks the beginning of the main program. This is the point at which execution will begin. A typical program structure would be:

```
<<Variable & Function Declarations >>>
```



```
Begin Program
        <<Statements >>
End Program
<<More Variable & Function Declarations >>
```

See Also: Program Layout

Calibrate Joystick

Calibrate Joystick *?JoyStickNo?, ?DeadBandX?, ?DeadBandY?*

Calibrate joystick is used to set the centre point of the specified joystick by providing a deadband, specified as a % around the location of the stick at the time of execution of this statement. All movement is then scaled proportionally so reading a joystick will provide a smooth linear progression from (Center) 0 – (Full throw) 100 representing the distance from Centre

The three arguments to the statement are all byte values – literal or constants. Specifying a variable will cause a compiler error.

Example:

```
Calibrate Joystick 1,10,10        ;Picks up current stick position and wraps a 10%
                                  ;dead band around that point – which is considered centre
```

See Also: ReadJoystick, ReadJoystickDir

Call

Call *?MyFunction?(?arg1?,?arg2?.....?argn?)*

Required method of invoking a user defined function that does not return a value.

Call MyFunction(MyArgs)

See Also: Declare Function

ConfigStepper

ConfigStepper *?StepperNo?, ?MinWait?, ?MaxWait? , ?HoldPercent?*

ConfigStepper is used to set up the parameters used by the SetStepper command.

StepperNo – 1 or 2
MinWait – a number from 0 to 255 to wait between steps at the fastest speed. Set to the lowest value that will allow your stepper to turn and not lose steps.
MaxWait – a number from 0 to 255 to use when starting a stepper movement. SetStepper will automatically ramp up the speed to the maximum speed.
HoldPercent – a percentage from 0 to 99 for the amount of current used when the stepper is not turning, to hold its position. It is suggest that 10% is typically sufficient.

The four arguments to the statement are all byte values – literal or constants. Specifying a variable will cause a compiler error.

Example:

```
ConfigStepper 1,10,100,10
```


Decrement

Decrement *?Variable?*

Used to decrement the contents of the variable by one. Variable may be a Word, Byte, or DataPointer .

This is the recommended way of decrementing a variable as it is easy to read in the code and is more efficient in memory use and execution than the familiar

```
MyByte = MyByte - 1
```

See Also: Increment

Defer Events

Similar in behaviour to Disable Events, with one key difference. Any events that arrive while Defer Events is asserted are cached and will be processed as soon as the next Enable Events statement is executed. Note the max queue depth for any one event is 255. Defer events is designed to be used to protect a critical piece of code from interruption without loosing the interruption.

Example:

```
Defer Events

.....Do  some code here that cannot be interrupted.

Enable Events
```

See Also: Disable Events, SetEvent, RemoveEvent, RaiseEvent, Declare EventHandler, Enable Events

Disable Events

No Arguments. Halts the raising of all events. Recommended to be used before setting up or removing individual events watches

See Also: Defer Events, SetEvent, RemoveEvent, RaiseEvent, Declare EventHandler, Enable Events

DisplayChars

DisplayChars ?Char1?,?Char2?

Places the character corresponding to the ascii code of the supplied variable on each of the digits of the seven segment LED display. Where the ascii code corresponds to a non display character three horizontal bars will be displayed in that digit.

Example:

```
Declare Byte MyByte1 = 48

Declare Byte MyByte2 = 49

DisplayChars MyByte1, MyByte2     ;Will place “10” on the display

DisplayChars "G","O"              ;Will place “GO” on the display
```

See Also: DisplayHex, DisplayNumber

DisplayHex

DisplayHex *?ByteVar?*

Places the contents of the specified Byte variable onto the seven segment LED display on the Motorvator in Hex

format. The is useful for debugging and testing feedback as the two digit display can thus display the whole range of a byte variable 00 – 0FFH (0 – 255).

See Also: DisplayChars, DisplayNumber

DisplayNumber

DisplayNumber *?ByteVar?*

Places the contents of the specified Byte variable onto the seven segment LED display on the Motorvator in Decimal format. The is useful for debugging and testing feedback but note the two digits will only display the two least significant digits, so values over 100 will appear without the leftmost (100 or 200) digit.

See Also: DisplayHex, DisplayChars

EmergencyStop

No Arguments. Stops all motors, Turns off all Action ports, Resets the servos to their central position and stops the current tune playing.

See Also: Escape Program

Enable Events

Back to Index

No Arguments. Enables system events to percolate through to your code where **Set Event Watch** code has been implemented. Note any events configured and working

before or since the corresponding Disable Events statement was executed will re-awaken.

See Also: Defer Events, Disable Events, SetEvent, RemoveEvent, RaiseEvent, Declare EventHandler

End Program

Marks the formal end of the main program. When execution of the code reaches this point the Motorvator will stop program execution and return to a ready state awaiting further commands. See Begin Program for more information on layout and use.

See Also: Program Layout, Escape Program, PowerOff

Escape Program

Back to Index

No Arguments. Terminates the current program, but does not alter any current device settings. Therefore any motors going, servos positioned or action ports switched on will remain so. This statement is useful for debugging, or when you want to leave a servo in a fixed position.

See Also: Emergency Stop, Program Layout, End Program, PowerOff

Include File

Include File "?FileName?"

Directs the compiler to include the specified filename as part of the program being compiled. The file will be appended to the main program as if it was part of the main file at the point the include statement was discovered and must follow overall rules for that point in the program. For example if the included file contains variable declarations that can only

be global , (such as DataTables), and is included within a function declaration then it will generate an error. Also note Functions cannot be declared within other functions.

Include files must also observe variable declaration rules – ie no two global or local variables can have the same name. Function names, by definition are global so therefore an include file cannot contain a function declaration identical to the main program or any other includes.

Includes can be nested to a maximum of 255 levels – that is a file that is included can contain include directives within itself. Careful the included files do not reference files already included as this will create a circular reference and will generate an error.

?Filename? must be the filename and extension of the file to be included. This will be appended to the compile path. Therefore you may use .\ and ..\ to provide a relative path instead of absolute. If the filename includes the Colon (:) character then it is assumed to reference an absolute filename and this will be used as-is and the filename will not be appended to the default pathname.

Files can be of any text type provided it obeys syntax rules. Thus a CSV file (A comma delimited format that Excel is capable of producing), could be maintained externally, yet imported and compiled successfully between Declare DataTable and End DataTable statements which would make for a tidy layout.

See Also: Program Layout

Increment

Increment *?Variable?*

Used to decrement the contents of the variable by one. Variable may be a Word, Byte or DataPointer .

This is the recommended way of incrementing a variable as it is easy to read in the code and is more efficient in memory use and execution than the familiar

MyByte = Mybyte + 1

See Also: Decrement

PowerOff

No Arguments. Turns off power to peripheral chips and stops all devices similar to emergency stop. CPU enters a sleep mode and awaits the pressing of the power on button.

See Also: *Escape Program*

Push

Push *?ByteVar? /OR/ ?WordVar? /OR/ ?BooleanVar? /OR/ ?DataPointer?*

Push places the specified variables value onto the user program stack where it can be accessed later with a corresponding pop statement.

CAUTION: The stack is used to manage variables through the user function call process and to protect the content of local variables – especially in the case of recursive code. Pushing or popping data values on and off the stack will interfere with this process and could create a "difficult to find" bug. As a rule all pushes should be matched with a corresponding pop within the same code scope. That is – within the main program, or within the same function declaration.

See Also: *Pop*

RaiseEvent

RaiseEvent *?EventType?*

Allows your code to explicitly raise an event. Especially useful for the 8 User Defined events, but can be used for debugging or simulating the system related events such as BateryVoltageEvent

See Events Constants for a list of events that can be raised

See Also: Defer Events, Disable Events, SetEvent, RemoveEvent, Declare EventHandler,Enable Events, Events Constants

Receive / 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
07308408804804905005105205 3<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 0730030000000000000000
; every 5 seconds it will display
07308408803202108208803 2001 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 1,610      ; and set up to send again in 5
seconds (5 x 122 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 and transmit if there is already a transmission in progress. This is why the TransmitEvent is provided, to indicate when the last transmission has completed.

RemoveEvent

RemoveEvent *?EventType?*

Ceases monitoring of the specified event type.

See Also: Defer Events, Disable Events, SetEvent, RaiseEvent, Declare EventHandler, Enable Events, Events Constants

SetAction

SetAction ?PortNo?, ?ByteValue?

Sets the specified Action Port (1-3) to the specified value. Both arguments to the statement are byte values – literals or constants. Specifying a variable will cause a compiler error.

?ByteValue? can be either 0 (turn Action Port OFF) or 1 (Turn Action port ON)

Example:

```
SetAction 1, 1                  ;Sets the first action port
to on.
```

SetEvent

SetEvent *?EventType?, ?UserFunctionName?*

Sets your code up to handle a system event. ?EventType? Must be one of the system defined event constants, while the ?UserFunctionName? must be the name of a user function that takes no arguments and returns no values.

If the event is already set to another location, it will be redirected to the one specified here.

Example:

```
Begin Program

        Disable Events                          ;Good
practice to disable

        SetEvent BatteryVoltageEvent, LowJuice
;while changing

        Enable Events

End Program

Declare EventHandler LowJuice()

        << Code to handle the event >>

End Function
```

See Also: Defer Events, Disable Events, SetEvent, RemoveEvent, RaiseEvent, Declare EventHandler, Enable Events, Events Constants

SetMotor

SetMotor *?MotorNo?,?Direction?,?Speed?*

SetMotor sets the specified Motor (1-4) to the specified direction ("F" for forward, or "B" for backward) and speed

(0-100). All three arguments may be literals, constants or byte variables, which aids writing common driver code to cope with multiple axes of movement.

Some Examples:

```
SetMotor 1,Forward,100          ;Forward is a system
constant for F

SetMotor MyMotor, MyDirection, MySpeed

SetMotor  2,"F",0                ;Stops motor two
```

See Also: StopMotor, StopMotors

SetServo, SetServo%

SetServo *?ServoNo?, ?Direction?, ?Degrees?*

SetServo% *?ServoNo?, ?Direction?, ?Percentage?*

Both Statements set the specified servo to the position in the specified direction. One accepts a variable containing degrees (0-90) while the other accepts a byte Variable contain 0-100 %. Depending on your coding situation, one will suit better than the other.

The SetServo% is useful if you want to take the result of a ReadJoystick() and apply it directly to the servo, e.g.

```
Angle = ReadJoystick(1,"X")

Direction = ReadJoystickDir(1,"X")

If Direction = Forward Then

   SetServo%(1,Forward,Angle)

else
```



```
   SetServo%(1,Backward,Angle)

end if
```

ServoNo must be a byte literal or constant 1 or 2

Direction must be a byte literal or Constant for "F" (Forward) or "B" (Backward)

SetStepper

SetStepper *?StepperNo?, ?Direction?, ?Steps?*

Direct the stepper motor to turn in ?Direction? for ?Steps? number of steps.

The MotorVator will manage the ramp up and slow down of the stepper speed.

The EVENT_STEPPER1_STOP and EVENT_STEPPER2_STOP events can be set to trigger when the stepper has finished moving (so you can go onto the next movement).

StopMotor

StopMotor *?MotorNo?*

Stops the specified motor. ?MotorNo? must be a literal value or constant between 1 and 4. Variable names will cause a compiler error.

See Also: SetMotor, StopMotors

StopMotors

No Arguments. Stops all motors.

See Also: SetMotor, StopMotor

StopTune

No Arguments. Stops the current tune playing.

See Also: Beep, PlayTune

Transmit

See Receive

WriteData

WriteData *?MyDataPointer?, ?WordVar? /OR/ ?ByteVar? /OR/ ?BooleanVar?*

WriteData is used to place data into a predefined DataTable via the specified data pointer. The provided data (Word, Byte or Boolean) is written to the current address pointed to by the Data Pointer, and the pointer incremented to the next free position. NB No checks are performed to ensure the pointer is in fact writing within the bounds of a declared data table, so this command should be used with utmost care.

Example:

```
Declare DataTable  MyTable

      Reserve 10
```

```
End DataTable

Declare DataPointer MyPointer

Declare Byte MyByte = 99

MyPointer = SetPointer( MyTable)

WriteData MyPointer, MyByte                ;Writes 99
                                           into the first
                                           location of
                                           MyTable
```

See Also: ReadData, SetPointer

PlayTune

Playtune ?DataTableName?

The contents of the DataTable must follow the modified RTTTL (Ring Tone Transfer Language) format rules.Each note has the following format:
[1,2,4,8, or 16 for the duration (def = 4)] optional
A to G for the Note (P for silent note)
[.] optional dot to extend duration by 1/2
[# for sharp] optional
[5 or 6 for the octave] optional (def = 5) 4C#5 is a 1/4 note of lower c#
2C#6 is a 1/2 note of upper c#
C is 1/4 tone of C in lower octave
Notes are separated by Commas. E.g. the startup fanfare is 16C,16E,16G,8C6,16G,2C6
The differences from full RTTTL

- No spaces allowed in string
- No tune name or parameter commands
- MotorVator only supports octaves 5 and 6
- Note Names must be in CAPITALS

The DataTable must end with either a 13 (in which case the tune will finish) , or a "<" (in which case the tune will repeat, until StopTune is called or another PlayTune is started).

For example, to create a "Truck Reversing Beep"

```
Declare DataTable    TruckReversing
          "C6,P<"
End Datatable
.....
SetMotor    1,Backward,Speed
SetMotor    2,Backward,Speed
PlayTune TruckReversing          ' will keep beeping until
stopped
Wait 220                                 ' go back for about 2
seconds
StopMotors
StopTune
```

See Also: Beep, StopTune

Return

Return *?OptionalVarToReturn?*

Return is used within function declarations to specify the specific value or contents of the specified variable to be returned to the calling code. The type of the variable specified must match what is declared in the function declaration. That is; if the function is declared as returning a byte, then attempting to return a word will generate a compile error.

Example:


```
Declare Function MyFunction(MyArgument as Byte) Returns
Byte

        If MyArgument = 0 then

                Return 0

        Else

                Return MyArgument

        End If

End Function
```

Wait

Wait ?TimeToWait?

Waits for the specified time in approximately 110ths of a second supplied by the word variable, literal or constant represented by ?TimeToWait?

Example:

```
   Wait 110          ;Waits approximately 1 second before
continuing
```

Note that during a WAIT operation, no other instruction (including Events) can run. Therefore, be wary of using long waits, and you delay an important Event change.

It is better to use the Timer Events, or to use a loop with a counter (or even to use a short WAIT 1 within a loop).

Math Handling

MeccCode supports simple math statements as follows in the rest of this section. Each step must have a variable to assign the value to, and up to two arguments and an operator. The arguments can be variables, literals or constants.

Addition

?MyByte? = ?A? + ? B?

?MyWord? = ?A? + ?B?

?MyPointer? = ?A? + ?B?

?A? and *?B?* can be Variables, Literals or constants that resolve to values.

Assignment

?MyByte? = ?A?

?MyWord? = ?A?

?A? can be Variables, Literals or constants that resolve to values.

Division

?MyByte? = ?A? / ? B?

?MyWord? = ?A? / ?B?

?MyPointer? = *?A?* / *?B?*

?A? and *?B?* can be Variables, Literals or constants that resolve to values.

Only a whole number result is returned. Use MOD to obtain the remainder. Divide by zero results in a zero result.

Subtraction

?MyByte? = *?A?* - *? B?*

?MyWord? = *?A?* - *?B?*

?MyPointer? = *?A?* - *?B?*

?A? and *?B?* can be Variables, Literals or constants that resolve to values.

Multiplication

?MyWord? = *?A?* * *?B?*

?MyPointer? = *?A?* * *?B?*

?A? and *?B?* can only be Byte Variables, Literals or constants that resolve to Byte values.

Note the sum is returned in a word variable. Word multiplication is not supported.

Mod

?MyByte? = *?A?* **MOD** *? B?*

?MyWord? = *?A?* **MOD** *?B?*

?MyPointer? = *?A?* **MOD** *?B?*

?A? and *?B?* can be Variables, Literals or constants that resolve to values.

The result is set to the remainder of the division. This can be useful if you have a loop that needs to update the screen every 50 times through as illustrated in this example:

Example:

```
Declare Byte Counter = 0

Declare Byte Check = 0


Do

        Counter = Counter + 1

        <Some Statements that do the work>

        Check = Counter MOD 50      ;If returns 0 then counter is divisible by 50

        If Check = 0 then

                DisplayNumber Counter    ;Update display every 50 cycles

        End If

Until Counter = 1000                    ;Want to do the loop 1000 times
```

And

Does a logical (bitwise) AND between the specified values

?MyByte? = *?A?* **AND** *? B?*

?MyWord? = *?A?* **AND** *?B?*

?MyPointer? = *?A?* **AND** *?B?*

?A? and *?B?* can be Variables, Literals or constants that resolve to values.

To understand bitwise AND study the following **example**

```
Declare Byte OddsOn = 0AAh     ;In Binary = 10101010

Declare Byte NibbleOn = 0Fh     ;in Binary = 00001111

Result = OddsOn And NibbleOn       ;In Binary = 00001010
```

Therefore if the corresponding bit in one byte AND the other is on then the result is on in the result

Or

Does a logical (bitwise) OR between the specified values

?MyByte? = *?A?* **OR** *? B?*

?MyWord? = *?A?* **OR** *?B?*

?MyWord? = *?A?* **OR** *?B?*

?A? and *?B?* can be Variables, Literals or constants that resolve to values.

To understand bitwise OR study the following **example**

```
Declare Byte OddsOn = 0AAh        ;In Binary = 10101010

Declare Byte NibbleOn = 0Fh        ;in Binary = 00001111
```



```
Result = OddsOn Or NibbleOn          ;In Binary = 10101111
```

Therefore if the corresponding bit in one byte OR the other is on then the result is on in the result

Code Looping & Conditional Structures

This section outlines the code constructs used to force execution to change its execution point depending on the result of a specified condition. The change in execution can be to selectively process a group of instructions (See IF and Select Case Structures) or to loop around to run a set of instructions many times (See Do, While, Loop Structures).

Condition syntax

The acceptable conditions for all conditional structure statements are as follows:

?MyVa1a? = *?MyVa1b?*

/OR/ *?MyVa1a?* > *?MyVa1b?*

/OR/ *?MyVa1a?* < *?MyVa1b?*

/OR/ *?MyVa1a?* <= *?MyVa1b?*

/OR/ *?MyVa1a?* >= *?MyVa1b?*

/OR/ *?MyVa1a?* <> *?MyVa1b?*

/OR/ Not(*?MyVa1a?*) [[= TRUE] /OR/ [= FALSE]]

/OR/ *?MyFunction?*([*?Args?*]) *?*) [[= TRUE] /OR/ [= FALSE]]

?MyVala? must be a variable and ?MyValb? can be a Variable, Literal or Constant of type Byte, Boolean, Word or DataPointer, but they must have the same type. That is you cannot match a word to a byte variable.

If you are matching a variable to a literal or you must place the variable on the left. For example:

```
If MyVar >= 6 then

If MyLoopCounter = MAX_TRIES then
                    ;MAX_TRIES has been declared as
                    constant
```

The system function Not() is the only system function that can be used as a condition.

You can use a function you create yourself as a condition – provided it returns a type of Boolean. Any other type return will generate an error at prime time. Note also the = True or = False is entirely optional, but may make your code more readable

Example:

```
    If LimitSwitchClosed(SwitchNo) Then
        <do something here >
    Else
        <Do something else>
    End If

Declare Function LimitSwitchClosed(SwNo as Byte) Returns
Boolean
    <Do something here>
End Function
```


Do / Until

Used where a conditional loop is needed – and the condition should be checked at the bottom of the loop. This means at least one pass through the loop will be completed before the condition is checked.

Example

```
Do
    <Statements>
Until ?Condition? [= True]
```

If the condition evaluates true then execution restarts with the first instruction after the DO statement

Loop / End Loop

Used where a loop is needed that will go on indefinitely – that is – there is no condition to evaluate. This is useful for the main loop of the program or testing hardware.

Example:

```
Loop
    <Statements are executed forever>
End Loop
```


While / End While

Used where a conditional loop is needed – and the condition should be checked at the top of the loop. This means the code within the While...End While section will only be executed while the condition evaluates as true.

Example:

```
While ?Condition? [= True]
    <Statements executed while the condition is true>
End While
```

If Then / Else / End If

Used to execute a group of instructions conditionally. This is not a loop structure.

Example:

```
If ?Condition? Then
    <Statements to execute if condition was true>

End If
```

When used with the optional Else statement – you get two blocks of statements – either one OR the other will be executed depending on the condition

Example:

```
If ?Condition? Then
    <Statements to execute if condition was true>
Else
    <Statements to execute if condition was false>
End If
```

If Structures can be nested as many levels deep as you like:

Example:

```
If ?Condition? Then
    If ?Condition? Then
        If ?Condition? Then
            <Statements to execute if condition
        was true>
        Else
            <Statements to execute if condition
        was false>
        End If
    Else
        <Statements to execute if condition was
    false>
    End If
```

```
Else

        If  ?Condition? Then

                If  ?Condition? Then

                        <Statements to execute if condition
        was true>

                Else

                        <Statements to execute if condition
                was false>

                End If

        End If

End If
```

Indentation is not required, but highly recommended as it is otherwise easy to omit a matching end if and the resulting complier error can be difficult to locate. If you are planning a complex nested if statement – consider the Select Case – it may make your code easier to maintain.

Select Case

Used where a variable (Byte variable or word variable only) needs to be compared to a number of different values and action taken accordingly. Once the corresponding block of statements are executed execution recommences with the first valid instruction after end Select. Select Case statements can be infinitely nested.

Example:

```
        Select Case arg1B

                Case 1
```



```
                        <Statements Only
                         executed if arg1B  1>

                Case MyByte

                        <Statements Only executed if arg1B =
                        MyByte>

                Case 3,4,5

                        <Statements Only executed if
                         arg1B = 3 OR 4 OR 5>

                Case Else

                        <Statements Only executed if arg1B
                        does not equal any other value in the
                        list>

        End Select
```

Predefined System Constants

True

In MeccCode, True is defined as not Zero. That is a byte of any value except 0. True as a constant is defined as 255 decimal.

Can be used in Math or Boolean functions

For Example:

```
MyByteVariable = True
```

Or

```
If MyVar =  False Then
```


False

False is defined as 0. Can be used in Math or Boolean functions as per the example in constant 'True' above.

Forward

Defined as the single ascii character "F". Can be used to make some commands more readable.

Example:

```
SetMotor 1, Forward, 50
```

Backward

Defined as the single ascii character "B". Can be used as per the constant 'Forward'

Example:

```
SetServo 1, Backward, 40)
```

UpButton

Value = 5. Used with ReadButton to extract the current value of the up button

DownButton

Value = 6. Used with ReadButton to extract the current value of the up button

Events Constants

These constants are designed to be used in the various events statements and functions were event type is required.

Event Constant	Value
EmergencyStopEvent	3
ButtonOnEvent	4
ButtonUpEvent	5
ButtonDownEvent	6
EVENT_STEPPER1_STOP	7
EVENT_STEPPER2_STOP	8
ReceiveEvent	9
TransmitEvent	10
OnOff1Event	11
OnOff2Event	12
OnOff3Event	13
OnOff4Event	14
BatteryVoltageEvent	15
Timed1Event	16
Timed2Event	17
Timed3Event	18
Timed4Event	19
User1Event	20
User2Event	21
User3Event	22
User4Event	23
User5Event	24
User6Event	25
UserDefined7Event	26
UserDefined8Event	27

See Also: Defer Events, Disable Events, SetEvent, RemoveEvent, RaiseEvent, Declare EventHandler, Enable Events

Compiler Option Switches

Option Force Globals

Option Force Globals [[On] /OR/ [Off]]

By default the compiler treats all variables declared at the top of the program as globals and all variables declared within function declarations as local variables. This requires a surprising number of additional instructions when the program is compiled and run, so this option allows you to force all variables into globals and results in a faster and smaller program.

Program Layout

Programs are expected to be set out in the following manner. A template will be added to the Environment that will be invoked each time 'New File' is selected that sets out the basic structure:

Example Program Layout

```
;Compiler directives go here (recommended location)

;Declare global variables here  (recommended location)

;Declare Functions and Event Handlers can go here
```



```
Begin Program

    ; Main Code Body statements here

End Program

   ;Can Declare global variables here

   ;Declare Functions and Event Handlers can go here (recommended location)
```

Layout rules are as follows. If these rules are not observed an error is generated and compile halted.

1. Begin & End Program must exist in all Programs and be present only once.
2. Commands cannot exist outside of Begin & End Program unless they are part of a Function Declaration or Event Handler.
3. Declare .... Statements cannot exist between Begin & End Program statements, but can go either before or afterwards.
4. Compiler directives must be placed before the Begin Program statement except the Import Directive which can exist anywhere in the program file.
5. Include file directive can go anywhere, but what is being included must fit in with the overall structure.

Appendix A: AWOS Instructions

The Motorvator uses AWOS instructions, which operate at a lower level than your MeccCompiler instructions. When you compile a MeccCompiler instruction, it may generate between one and three lines of AWOS instructions. This section is provided because when you are Debugging your code under the CLI debugger (see page 27), these are the instructions that you will see being executed by the MotorVator. The bracketed numbers (e.g.the #54 in Add Byte (54)) allow you to match the instructions.

Argument Types

In the following instructions, you will see that each instruction has ARGUMENTS. These arguments are the values that the instruction uses. So an ADD BYTE instruction does a C=A+B function and so needs three arguments (A,B,C). The type of arguments varies:

Argument Type	Means
Byte Variable	The of a variable that you've declared As Byte
Byte Constant	A fixed byte value (number from 0-255, single Character).
Word Variable	The name of a variable that you've declared As Word
Label	The Address of the next piece of code to execute

Hardware Active Instructions.

Hardware Input Instructions.

Program Flow Instructions.

Data Instructions.

Event Instructions.

Add Byte (54)

Argument	A	B	C
Type	Byte variable or Constant	Byte variable or Constant	Byte Variable

see also: Add Word, Subtract Byte

C = A+B

Adds Byte A to Byte B and puts result in Byte C

Add Word (62)

Argument	A	B	C
Type	Word variable or constant	Word variable or constant	Word variable

See Also: Add Byte, Subtract Word

C = A+B

Adds Word B to Word A and puts result in Word C

Calibrate Joystick (29)

Argument	Joystick no	Deadband X %	Deadband y%
Type	Byte Value	Byte Value	Byte Value

The Calibrate Joystick instruction sets up Joystick one or Joystick two to provide Adjusted Readings.

Call (51)

Argument	A		
Type	Label		

Call Code Subroutine A, i.e. Start Execution of code at Label A

Execution will return to the next statement (after this Call) when a RETURN is executed.

Clear Byte (61)

Argument	A		
Type	Byte Variable		

Set the Value of Byte variable A to be ZERO.

A = 0

Clear Word (66)

Argument	A		
Type	Word Variable		

Set the Value of Word Variable A to be ZERO.

A = 0

Complete Event (75)

Argument	No arguments		
Type			

Compliment Byte (60)

Argument	A		
Type	Byte Variable		

Compliment (swap all the bits over from 1 to 0 and vice versa within the byte).

Use also for swapping a FLAG variable from TRUE to FALSE or vice versa.

Convert Byte (42)

Argument	A	B	
Type	Byte Variable	Word Variable	

Convert Word (43)

Argument	A	BHi	Blo
Type	Word Variable	Byte Variable	Byte Variable

Use to convert a Word Variable into two Byte variables.

Decrement Byte (65)

Argument	A		
Type	Byte Variable		

Decrement the value of the Byte Variable A by 1.

A = A - 1

Decrement Word (65)

Argument	A		
Type	Word Variable		

Decrement the value of the Word Variable A by 1.

A = A - 1

Defer Events (22)

See Also: Enable Events

No Arguments

After this command, no Events will interrupt your program.

The events will still be logged, but the Event Code established by a Set Event Watch will be temporarily disabled.

To re-activate the events, issue an ENABLE EVENTS instruction.

The difference between *Defer Events* and *Disable Events* is that the events will still be logged/counted under a *Defer Events*, and will be available once the *Enable Events* is issued. A Disable Events removes the monitoring of events completely.

For example, if you issue a Defer Events, and three event triggers happen before the Enable Events is issued, then the Event Code will be run three times before your program continues.

Disable Events (6)

See Also: Enable Events

No Arguments

After this command, no Events will be active. If you are using events, if often pays to start with a Disable Event, then any SET EVENT WATCH, then any other initialisations, then before the main program starts, issue an ENABLE EVENTS instruction. This stops any events 'firing' before you are ready.

You can also Disable Events if you are in a time-critical part of your program that you don't want interrupted.

Display Hex (38)

Argument	A		
Type	Byte Variable		

Display the two character hexadecimal (base 16) representation of a single byte value, on the MotorVator two character display.

Display Number (39)

Argument	A		
Type	Byte Variable		

Display the two character decimal representation of a single byte variable, on the MotorVator two character display.

A single byte can contain a value from 0 to 255 (decimal).

NOTE: If the value is greater than or equal to 100, then the MotorVator will display the two least significant digits. I.e. the display is the same ('23") for 23 and 123 and 223, and there is no warning given. Only use if you are certain that your value is < 100, otherwise use Display Hex.

Divide Byte (46)

Argument	A	B	
Type	Byte Variable	Byte variable	

Divide A by B, putting the result into A and the remainder into B.

Note that the Divide Byte (and Divide Word) overwrites the values of both A and B, so MeccCompilerII uses temporary variables.

Divide Word (47)

Argument	A	B	
Type	Word Variable	Word Variable	

See Also: Divide Byte

Divide A by B, putting the result into A and the remainder into B. Use Divide Word (rather than Divide Byte) if the size of the calculations will exceed 255.

Note that the Divide Word (and Divide Byte) overwrites the values of both A and B, so MeccCompiler II uses temporary variables to keep the original values for later instructions.

Emergency Stop (1)

see also: Stop Motors

Stops all motors, turns off all Action Ports, Disables all Servos.

Enable Events (7)

See Also: Disable Events

Sets any Listed events (as defined using Set Event Watch instructions) to active.

End (5)

see also Escape, Emergency Stop

Your program will stop executing and return the MotorVator to the ON state.

Note that this statement WILL automatically reset/stop any Motors, Servos or Action Ports that you might have been using.

Escape (3)

see also End, Emergency Stop

Your program will stop executing and return the MotorVator to the ON state.

Note that this statement WILL NOT reset any Motors, Servos or Action Ports that you might have been using. This is so that you can leave specific ports, servos etc in a known state (e.g. you can leave a crane grab in the closed position, or leave a brake setting on).

Given that any motors etc running will not be stopped, it is up to you to ensure that all outputs are in a known state before using the Escape command.

Unless you're absolutely sure, use the END command.

Get Clock Ticks (67)

Argument	A		
Type	Word Variable		

Read s the current value of the System Clock Ticker into a word variable.

Get Program Number

Argument	A		
Type	Byte Variable or constant		

Get Program Number Which_Program

Returns the number that was selected on the MotorVator when starting the program.

If Byte Equal (80)

Argument	A	B	C
Type	Byte Variable or constant	Byte variable or constant	Label

If (A = B) then GOTO C

If Byte Greater Than (83)

Argument	A	B	C
Type	Byte Variable	Byte variable or constant	Label

If (A > B) then Jump to C

If Byte Greater Than or Equal (85)

Argument	A	B	C
Type	Byte Variable	Byte variable or constant	Label

If (A >= B) then Jump to C

If Byte Less Than (82)

Argument	A	B	C
Type	Byte Variable	Byte variable or constant	Label

If (A < B) then Jump to C

If Byte Less Than or Equal (90)

Argument	A	B	C
Type	Byte Variable	Byte variable or constant	Label

If (A <= B) then Jump to C

If Byte Not Equal (81)

Argument	A	B	C
Type	Byte Variable	Byte variable or constant	Label

If (A <> B) then Jump to C

If Byte Flag False (77)

Argument	A	B	
Type	Byte Variable	Label	

If not(A) then Jump to B

If Byte Flag True (76)

Argument	A	B	
Type	Byte Variable	Label	

If (A) then Jump to B

If Word Equal (86)

Argument	A	B	C
Type	Word Variable	Word Variable or constant	Label

If (A = B) then Jump to C

If Word Greater Than (89)

Argument	A	B	C
Type	Word Variable	Word variable or constant	Label

If (A > B) then Jump to C

If Word Greater Than or Equal (91)

Argument	A	B	C
Type	Word Variable	Word variable or constant	Label

If (A >= B) then Jump to C

If Word Less Than (88)

Argument	A	B	C
Type	Word Variable	Word Variable or constant	Label

If (A < B) then Jump to C

If Word Less Than or Equal (90)

Argument	A	B	C
Type	Byte Variable	Byte variable or constant	Label

If (A <= B) then Jump to C

If Word Not Equal (87)

Argument	A	B	C
Type	Word Variable	Word variable or constant	Label

If (A <> B) then Jump to C

Increment Byte (56)

Argument	A		
Type	Byte Variable		

Increment the value of the Byte Variable A by 1.

A = A + 1

Increment Word (64)

Argument	A		
Type	Word Variable		

Increment the value of the Word Variable A by 1.

A = A + 1

Jump (53)

Argument	A		
Type	Label		

Unconditional Jump

Cause execution to occur from Label A.

Logical AND (95)

Argument	A	B	C
Type	Byte Variable or constant	Byte Variable or constant	Byte Variable

Performs a Binary AND function between Byte Variable A and Byte Variable B, and puts the Result into variable C.

A Binary AND takes two bits with the following results:

AND	0	1
0	0	0
1	0	1

Use the AND command to extract specific bits from a byte, or to Turn OFF specific bits.

Logical OR (96)

Argument	A	B	C
Type	Byte Variable or constant	Byte Variable or constant	Byte Variable

Performs a Binary OR function between Byte Variable A and Byte Variable B, and puts the Result into variable C.

A Binary OR takes two bits with the following results:

OR	0	1
0	0	1
1	1	1

Use the OR command to Turn ON specific bits from a byte.

Loop Until ZERO (50)

Argument	A	B	
Type	Word Variable	Label	

See also: Jump, If Byte, If Word

Decrements Word Variable A by 1, then checks to see if it is now Zero.

If it is Not Zero, then jump to Label B.

Move Byte (11)

Move Byte

Argument	A	B	
Type	Byte variable or Constant	Byte Variable	

Moves the value stored at A into the Byte Variable B

B = A

Move Word (12)

Argument	A	B	
Type	Word variable or Constant	Word Variable	

Moves the value stored at A into the Word Variable B

B = A

Multiply Byte (58)

Argument	A	B	C
Type	Variable or Constant Byte A	Variable or Constant Byte B	Word Variable C

Multiply (Unsigned) Byte Value A by Byte value B and place result into Word Variable C.

Note that the value will always be positive, and that the product of two bytes (0-255) requires a full word (0-65535).

NOP (0)

No Arguments

A No OPeration Instruction.

Use if you want to remove another instruction temporarily, but still keep all the addresses for following instructions the same.

On Error (71)

Argument	Error Routine	Return Code	Opcode Address
Type	Label	Byte Variable	Address

If the MeccCode interpreter in the MotorVator encounters a system, internal or code error, then program control can be sent to an error routine.

Play Sound (48)

Argument	Note Value	Duration	
Type	Variable or Constant Byte	Variable or Constant Byte	

Play a sound (tone of which is determined by the value of Note Value) for Duration x 1/10 seconds. The sound has a lower tone if the Note value is Larger.

Note that Play Sound does not use the Note System as for Play Tune.

Play Tune (49)

Argument	Tune String		
Type	String or Address		

Play Tune will play a tune following modified RTTTL (Ring Tone Transfer Language) format rules.

Randomize Byte (59)

Argument	A		
Type	Variable Byte		

Will load a pseudo-random value between 0 and 255 into the Byte Variable A.

Read Analog Input (32)

Argument	Analog Port	Read_Value	
Type	Constant Byte	Variable Byte	

Read the current analog value from the specified port (1..6) into the variable Read_Value.

Read Button (35)

Argument	Button No	A	
Type	Constant Byte	Byte Variable	

Checks the current status of the Director Buttons, and sets Variable A to either 0 if the button is released or 1 if the button is depressed.

Button Numbers are

1 = Director Left Button

2 = Director Left Stick

3 = Director Right Button

4 = Director Right Stick

Read Data Byte (16)

Argument	Data Pointer	Variable	
Type	Word Pointer	Byte Variable	

Read the value of the byte pointed to by Data Pointer, store into Variable and increment the Data Pointer to point to the next byte.

Read Data Word (18)

Argument	Data Pointer	Variable	
Type	Word Pointer	Word Variable	

Read the value of the Word pointed to by Data Pointer, store into Variable and increment the Data Pointer to point to the next ord.

Read Digital Input (31)

Argument	Digital Port No	A	
Type	Constant. Value 1 to 4	Byte Variable	

See Also: Read Button, Read Timed Input, Read Analog Input

Read the current state of any of the Digital Inputs.

Variable A will be set to either 0 (if Digital input is open) or 1 (if Closed).

Read Joystick One (27)

Read Joystick Two (28)

Argument	Joystick Port	Direction	Reading
Type	Constant Byte	Variable Byte	Variable Byte

Read the current Joystick "adjusted values" from the specified Joystick1 Axis ("X" or "Y") into the variables Direction and Reading.

Each value will be 0 to 100, representing a reading from Center (0) to Up or Full Down or Full Left to Full Right (100). Direction will be either "F" or "B".

Read Pulse Count Digital (36)

Read Pulse Count Timed (37)

Argument	Port Number	Count	
	Constant (1..4)	Byte Variable	

Read Timed Input (33)

Argument	Port Number	State	LastHighWidth
	Constant (1..4)	Byte Variable	Byte Variable

Reads the current value of the Timed Input Ports, and also returns the duration of the last High Pulse on the port (even if the port is currently Low).

Receive (14)

See Also Transmit

Returns a received packet of eight bytes of data from the Communications Port. Use in conjunction with the Transmit Instruction to have the MotorVator communicate with a PC or other MotorVators.

Remove Event Watch (73)

Removes an existing User Event Watch (as defined with Set Event Watch) from the active list. Once removed, the Event code will be executed, even if the Event now occurs. You can reenable to Event by using the Set Event Watch.

Reset (8)

No arguments

Reset the user program to run from the beginning.

Return (52)

No arguments

Set Action Port (30)

Argument	Action Port Number	Action State	
Type	Constant (1..3)	Constant or Variable Byte	

Set the Output of the Action Port to either

Off - if value of Action State is ZERO

On - for any other value of Action State

Set Data Pointer (15)

Argument	Data Pointer	Address	
Type	Word Pointer	Label	

Set up a Data Pointer from which to read Data.

Subsequent Read Data and Write Data instructions will read the data from the position pointed to by the Data Pointer, then automatically increment the pointer to point to the next data value.

Set Event Watch (72)

Argument	Event Type	Call Address	
Type	Byte value	Label	

The defined events are:

Event Number	Event
4	On Button Pushed
5	Up Button Pushed
6	Down Button Pushed
9	Packet Received on Communications Port
10	Packet has been sent on Comms Port
11	On/Off Input 1 has changed
12	On/Off Input 2 has changed
13	On/Off Input 3 has changed
14	On/Off Input 4 has changed
15	Battery Voltage Now below 5.8V
16	;Timed input 1 has changed
17	;Timed input 2 has changed
18	;Timed input 3 has changed
19	;Timed input 4 has changed
20	;User Defined Event 1
21	;User Defined Event 2
22	;User Defined Event 3
23	;User Defined Event 4
24	;User Defined Event 5
25	;User Defined Event 6
26	;User Defined Event 7
27	;User Defined Event 8

Set Left LED (44)

Argument	Character		
Type	Constant or Variable Byte		

Displays the Character onto the Left LED Display on the MotorVator.

Character is in the range of " "..."?","0".."9","A" to "Z" only (capital letters only supported).

Set Flag False (77)

Argument	Flag		
Type	Constant or Variable Byte		

Set the value of the FLAG variable to FALSE (Zero). Can then be used in an If Flag False or If Flag Truce test.

Set Flag True (76)

Argument	Flag		
Type	Constant or Variable Byte		

Set the value of the FLAG variable to True. Can then be used in an If Flag False or If Flag Truce test.

Set Motor (21)

Argument	Motor Number	Direction	Speed
Type	Constant	Constant	Byte Variable

Set the speed and direction of the Motor Ports A through D.

Motor Port A is Motor Number 1

Motor Port B is Motor Number 2

Motor Port C is Motor Number 3

Motor Port D is Motor Number 4

Each motor can be set for either Forward "F" or Back "B". Note however that the Concept of Forward and Back is arbitrary, and will relate to your model, gearing etc. If the Forward command makes your model go backwards, simply swap the two leads on the Motor Output Port.

Motor Speed is 0 to 100% of full power.

Set Motor Speed to 0 (in either direction) to stop the motor.

Set Right LED (45)

Argument	Character		
Type	Constant Byte		

Displays the Character onto the Left LED Display on the MotorVator.

Set Servo

Argument	Servo Number	Direction	Angle
	Constant	Constant	Variable Byte

Set the selected Servo to an angle between Back (Anticlockwise) 90 degrees and Forward (clockwise) 90 Degrees.

Set Servo Percent

Argument	Servo Number	Servo Direction	Percent of 90 degrees
Type	Constant	“F” or “B”	Variable

Set also Set Servo,

This command is included to allow you to take a reading from READ JOYSTICK and use the reading (which will be in the range 0 to 100) to set a Servo across its range (physically 90 degrees forward or backwards).

Sleep (4)

Puts the MotorVator into a low power mode until an event happens. Only use this if you your program at this point of its execution relies solely on Events to proceed. Otherwise your program will sleep here forever.

While in Sleep mode, the Off Key still operates.

Stop Motors (20)

No Arguments

Stops all Motors, regardless of their current state.

Does not affect Servos or Action Port settings.

Subtract Byte (55)

Argument	A	B	C
Type	Variable or Constant Byte	Variable or Constant Byte	Variable Byte

C = A - B

Subtracts the Value of B from the Value of A, and stores result in C.

Subtract Word (63)

Argument	A	B	C
Type	Variable or Constant Word	Variable or Constant Word	Variable Word

C = A - B

Subtracts the Value of Word B from the Value of Word A, and stores result in Word C.

Transmit (13)

Transmits a packet of eight bytes of data out of the Communications Port. Use in conjunction with the Receive Instruction to have the MotorVator communicate with a PC or other MotorVators.

Transmit can be used to send information (e.g. for logging or debugging) to the CLI.

The data is send as three character sets of ascii values, separated by commas, e.g. a buffer containing 1,2,4,8,16,32,64,128 will be sent as
073, 001,002,004,008,016,032,064,128.
The 073 represents an “I” character which is always sent on the front of a transmission.

Wait for Time (70)

Argument	Time To Wait		
Type	Constant or Variable Word		

Pauses execution of code for [Time to Wait] times approximately 1/111 seconds, e.g. to wait for one second use WAIT FOR TIME 111

Write Data Byte (17)

Argument	Data Pointer	Variable	
Type	Word Pointer	Byte Variable	

Write the value of the byte variable into the byte pointed to by Data Pointer, and increment the Data Pointer to point to the next byte.

Write Data Word (19)

Argument	Data Pointer	Variable	
Type	Word Pointer	Word Variable	

Write the value of the WORD variable into the WORD pointed to by Data Pointer, and increment the Data Pointer to point to the next WORD.

Example: Use to write data into a table. Note that we use Multiply Byte to get the index into a Word format and index by 2s (because a word is two bytes) so we can add to the base pointer to get to the correct table entry.