BASICMICRO

MCP Series Brushed DC Motor Controllers

MCP233
MCP236
MCP263
MCP266
MCP2163
MCP2126
MCP2166
MCP2206
MCP2128
MCP2168

User Manual

Firmware V1.1 and Newer
Hardware V2 and Newer
User Manual Revision 1.2

Contents

Firmware History

MCP is an actively maintained product. New firmware features will be available from time to time. The table below outlines key revisions that could affect the version of MCP you currently own.

Revision	Description
1.0.0	• Initial Public Release

Warnings

There are several warnings that should be noted before getting started. Damage can easily result if the motor controller is not properly wired. Injury can occur due to inadequate planning for emergency situations. Any time mechanical movement is involved the potential for injury is present. The following information can help avoid damage to the motor controller, connected devices and help reduce the potential for injury.

Disconnecting the negative power terminal is not the proper way to shut down a motor controller. Any connected I/O to MCP will create a ground loop and cause damage to the MCP and attached devices.

Brushed DC motors are generators when spun. A robot being pushed or coasting can create enough voltage to power MCPs logic intermittently creating an unsafe state. Always stop the motors before powering down MCP.

MCP has a minimum power requirement. Under heavy loads, without a logic battery and, brownouts can happen. This will cause erratic behavior. A logic battery should be used in these situations.

Never reverse the main battery wires, MCP will be permanently damaged.

Never disconnect the motors from MCP when under power. Damage will result.

1.0 Introduction

1.0.1 Motor Selection
When selecting a motor controller several factors should be considered. All DC brushed motors will have two current ratings, maximum stall current and continuous current. The most important rating is the stall current. Choose a model that can support the stall current of the motor selected to insure the motor can be driven properly without damage to the motor controller.

1.0.2 Stall Current
A motor at rest is in a stalled condition. This means during start up the motors stall current can be reached. The loading of the motor will determine how long maximum stall current is required. A motor that is required to start and stop or change directions rapidly even with a light load may still require maximum stall current often.

1.0.3 Running Current
The continuous current rating of a motor is the maximum current the motor can run without overheating and eventually failing. The continuous current rating on most motors is several times lower than the stall current rating of the motor. The continuous running current of the motor should not exceed the continuous current rating of the motor to prevent damage to the motor.

1.0.4 Wire Lengths
Wire lengths to the motors and from the battery should be kept as short as possible. Longer wires will create increased inductance which will produce undesirable effects such as electrical noise or increased current and voltage ripple in the DC Link capacitors. The power supply/battery wires must be as short as possible. They should also be sized appropriately for the amount of current being drawn. Increased inductance in the power source wires will increase the ripple current/voltage which can damage the DC Link capacitors on the board and/or causing voltage spikes over the rated voltage of the motor controller, leading to controller failure.

1.0.5 Run Away
During development of your project caution should be taken to avoid run away conditions. The motors should be mounted securely and allowed to rotate freely until properly setup. If the motor is embedded, ensure you have a safe and easy method to remove power from the controller as a fail safe.

1.0.6 Disconnect
To assure powering off in an emergency, a properly sized switch and/or contactor should be used. Also because the power may be disconnected at any time there should be a path for regeneration energy back to the battery even after the power has been disconnected. Use a power diode with proper ratings to provide a path across the switch/contactor and any fuse.

1.0.7 Encoders
MCP features dual quadrature encoders along with various other encoder options such as absolute. When wiring encoders make sure the direction of spin is the correct direction relative to the motor direction. Incorrect encoder connections can cause a run away state. Refer to the encoder section of this user manual for proper setup. In addition some encoders can cause excessive noise on the +5VDC rail of the controller. This excessive noise will cause unpredictable behavior in velocity and position control modes.

1.0.8 Power Sources

A battery or DC power supply can be used as the main power source for the MCP motor controller. When using a DC power supplies, motor deceleration can cause excessive regenerative voltages. Typically excessive regenerative voltage must be bleed off using a voltage clamp circuit. This can also be resolved by limiting motor deceleration. This will reduce the regenerative voltages.

The MCPs minimum and maximum voltage levels can be set to prevent some of these voltage spikes, however this will cause the motors to brake when slowing down in an attempt to reduce the over voltage spikes. This will also limit power output when accelerating motors or when the load changes to prevent undervoltage conditions.

1.0.9 Primary Power

The MCP can operate as a single power source motor controller. The MCP has a built in DC/DC switching regulator to power the logic. The DC/DC circuit also supplies a +5VDC rail to power user devices. When powering external devices from the controller, ensure the maximum current rating is not exceeded. This can cause MCP to suffer logic brownouts which will cause erratic behavior.

1.0.10 Secondary Power

The MCPs logic circuits can be powered from a secondary power source. The logic power source will also power the DC/DC switching regulator. The main and secondary power sources are feed through a diode circuit to the DC/DC switching regulator circuit. The higher voltage between the main and secondary power sources will power the DC/DC switching regulator. When the MCP is under heavy load and the voltage drops below the secondary battery source it will become the preferential power source for the DC/DC circuit.

1.0.11 Manual Voltage Settings

The minimum and maximum voltage can be set using the Ion Studio software or packet serial/ CANopen commands. Values can be set to any value between the boards minimum and maximum voltage limits. This is useful when using a power supply without using a voltage clamp. A minimum voltage just below the power supply voltage (2 to 3v below) will prevent the power supply voltage from dipping too low under heavy loads. A maximum voltage set just above the power supply voltage (2 to 3v above) will help protect the power supply and MCP from regenerative voltage spikes if an external voltage clamp circuit is not being used by braking the motors to dissipate the over voltage.

1.0.12 Minimum Power

Depending on the model of MCP there is a minimum main power requirement of at least 10V. Under heavy loads, if the logic is powered from the main battery, brownouts can happen. This can cause erratic behavior from the controller. If this is the case a separate logic battery should be used to power the logic of the motor controller.

1.0.13 Easy to use Libraries

Source code and Libraries are available on the Ion Motion Control website. This includes libraries for Arduino(C++), C# on Windows(.NET) or Linux(Mono) and Python(Raspberry Pi, Linux, OSX, etc).

1.1 Hardware Overview

1.1.1 Control Inputs

All digital input are 15V tolerant. The MCP outputs can interface to both 5V and 3.3V logic with no voltage translation required. The MCP inputs are internally current limited and voltage cliped to 3.3V. This method also protects the MCP I/O from damage.

The digital output pins can drive up to 3Amps at 40VDC. The CAN interface and RS232 pins are duplexed. When a duplex function is used the other is set to a high Z state. R/C pulse input, Analog, TTL and PWM can be generated from any microcontroller such as a Arduino or Raspberry Pi. The R/C Pulse in pins can be driven by any standard R/C radio receiver. There are several user configurable options depending on the device used to control the MCP. Ion Studio is required to configure the MCP from the USB port.

1.1.2 Encoder Inputs

The MCP provides inputs for dual encoders and +5VDC for powering attached encoders. When connecting the encoder ensure the leading channel of the encoder is connect to Channel A on the motor controller. Channel A is configured to increment the internal counters. Refer to the data sheet of the encoder you are using for channel direction. The encoders can be swapped and paired to either motor channel from Ion Studio.

1.1.3 Main Battery Screw Terminals

The main power input can be from 10VDC to 80VDC depending on the MCP model. The main battery connections are marked **+** and **-** near the main screw terminal. The plus (**+**) symbol marks the positive terminal and the negative (**-**) marks the negative terminal. The main battery wires should be as short as possible.

Do not reverse main battery wires. Roboclaw will be permanently damaged.

1.1.4 Main Battery Disconnect

The main battery should have a quick disconnect in case of a run away situation. The switch must be rated to handle the maximum current and voltage from the main battery. This will vary depending on the type of motors and or power source you are using. A typically solution would be an inexpensive contactor which can be sourced from sites like Ebay. A power diode rated for the maximum current the battery will deliver should be placed across the switch/contactor to provide a path back to the battery when disconnected while the motors are spinning. The diode will provide a path back to the battery for regenerative power even if the switch is opened.

1.1.5 Motor Screw Terminals

The motor screw terminals are marked with M1A / M1B for channel 1 and M2A / M2B for channel 2. For both motors to turn in the same direction the wiring of one motor should be reversed from the other in a typical differential drive robot. The motor and battery wires should be as short as possible. Long wires can increase the inductance and therefore increase potentially harmful voltage spikes.

1.2 Ion Studio Overview

1.2.1 Ion Studio Setup Utility

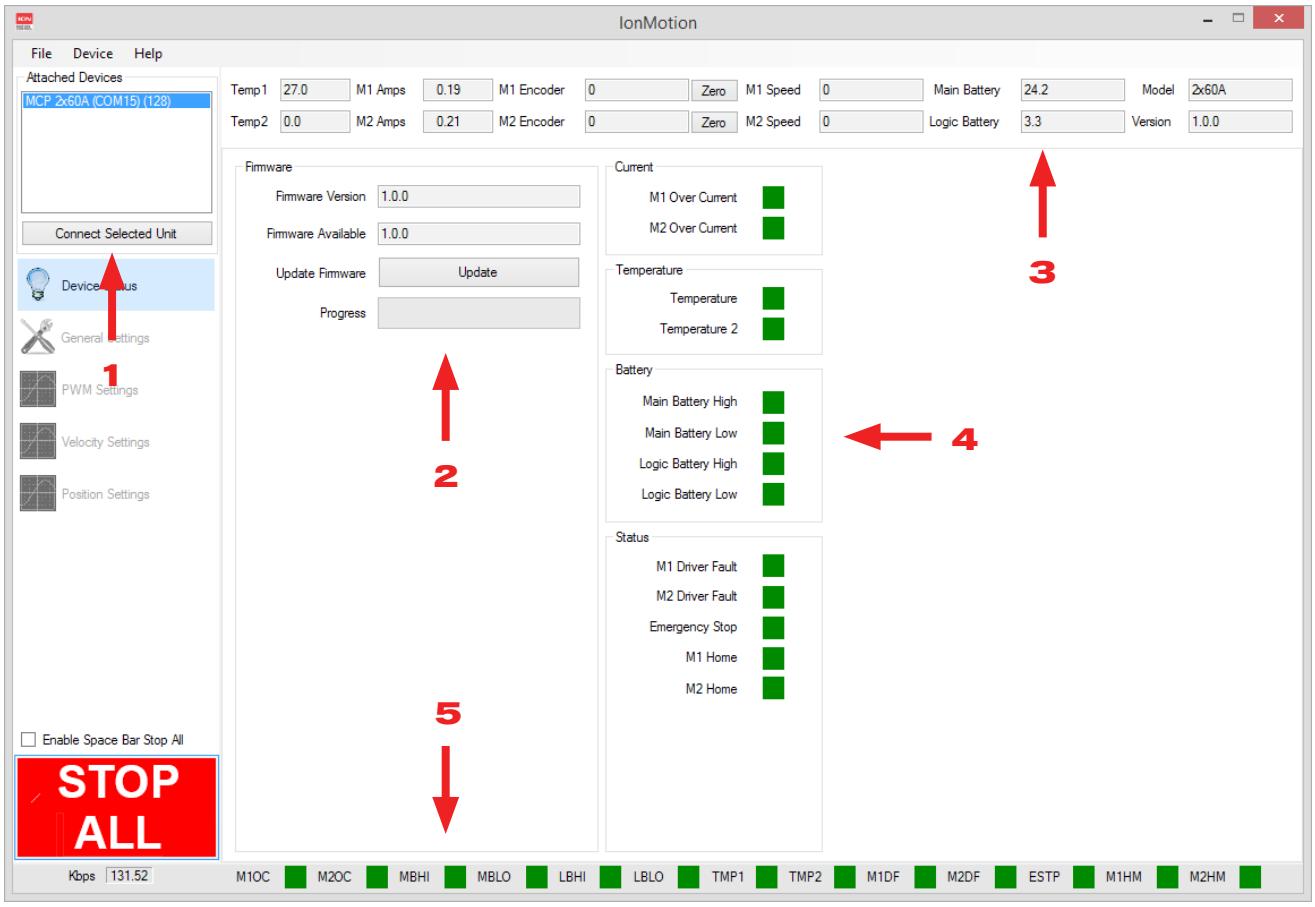
The Ion Studio software suite is designed to update firmware, configure I/O functions, configure MCP modes and create MCL programs. Ion Studio includes an editor for creating MCL programs to customize the MCP motor controller. The Ion Studio application can be downloaded from the ionmc.com downloads page.

1.2.2 Device Connection

This first screen shown is the Ion Studio connection screen. From this screen a detected motor controller must be selected (1). More than one motor controller can be detected. Only one can connect at a time from the selection box (1).

After the motor controller is connected it's firmware version is checked (2). If a newer firmware version is available it can be updated by clicking the Update Firmware button (2).

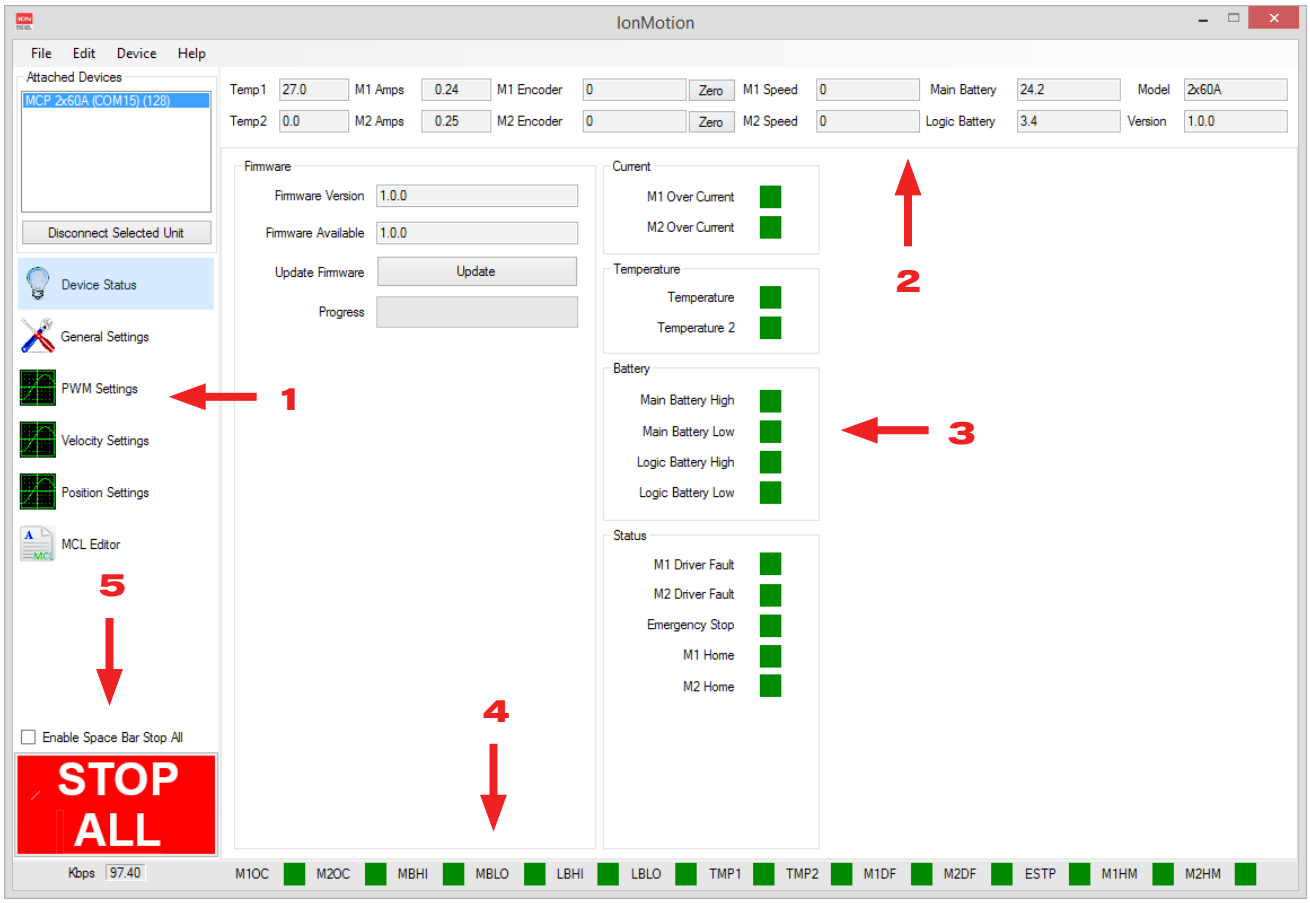
Fields (3,4,5) display current values and status. The fields at the top of the screen (3) show the current value for each of the monitored parameters and are updated live once a motor controller is connected. Status indicators (4) indicate the current condition of the monitored parameter. The bottom of the window includes a duplicate set of abbreviated indicators (5). These indicators (5) are shown on all screens. Green indicates the status is OK. Yellow is a warning. This typically means a limit is close to maximum. Red is an error. In most cases this will cause the unit to shut down the motors.



1.2.3 Device Status

Once a connection is established, the device status screen becomes active (1). All status indicators (3,4) and monitored parameter fields (2) will update to reflect the current status of the connected motor controller.

When a motor controller is connected the Stop All (5) button becomes active. There is a small check box to activate the Stop All function by using the space bar on an attached keyboard. This safety feature is the quickest method to stop all motor movements when using Ion Studio.



1.2.4 Device Status Screen Layout

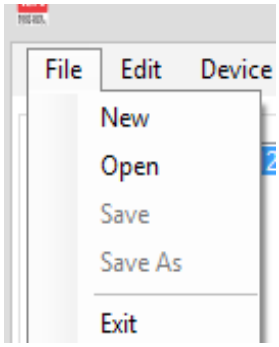
Label	Function	Description
1	Window Selection	Used to select which settings or testing screen is currently displayed.
2	Monitored Parameters	Displays continuously updated status parameters.
3	Status Indicators	Displays current warnings and faults.
4	Status Indicators	Displays abbreviated status of warnings and faults. Visible at all times.
5	Stop All	Stops all motion. Can be activated from keyboard space bar.

1.2.5 Status Indicator

The status indicators (4) shown at the bottom of the screen are an abbreviated duplication of the main status indicators (3) shown on the device status screen.

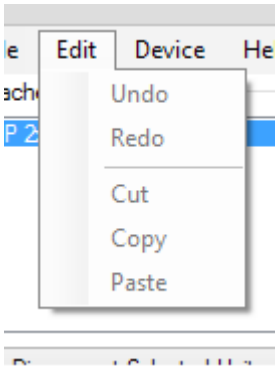
Label	Description
M1OC	Motor 1 over current.
M2OC	Motor 2 over current.
MBHI	Main battery over voltage.
MBLO	Main battery under voltage.
LBHI	Logic battery over voltage.
LBLO	Logic battery under voltage.
TMP1	Temperature 1
TMP2	Optional temperature 2 on some models.
M1DF	Motor driver 1 fault. Not applicable to MCP motor controllers.
M2DF	Motor driver 2 fault. Not applicable to MCP motor controllers.
ESTP	Emergency stop. When active.
M1HM	Motor 1 homed or limit switch active. When option in use.
M2HM	Motor 2 homed or limit switch active. When option in use.

1.2.6 File Menu



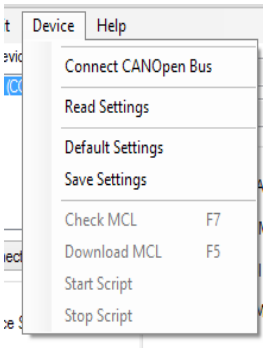
Label	Description
New	Create a new MCL program file.
Open	Open an existing MCL program file.
Save	Save an open MCL program file.
Save As	Save an open MCL progam file to a new filename.
Exit	Disconnects from the unit and close Ion Motion.

1.2.7 Edit Menu



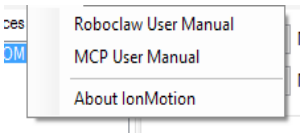
Label	Description
Undo	Undo last edit in MCL editor.
Redo	Redo last edit in MCL editor.
Cut	Cut selected text in MCL editor.
Copy	Copy selected text in MCL editor.
Paste	Paste copied text in MCL editor.

1.2.8 Device Menu



Label	Description
Connect CANOpen Bus	Connects or Disconnects from a GridConnect CANUSB Bus controller.
Read Settings	Reads the settings saved in non-volatile memory on the controller.
Default Settings	Loads the factory default settings to the controller.
Save Settings	Writes current settings to non-volatile memory on the controller.
Check MCL	Compiles and checks the open MCL script for errors.
Download MCL	Downloads the MCL script to the controllers flash memory.
Start Script	start currently programmed script
Stop Script	stop script if active

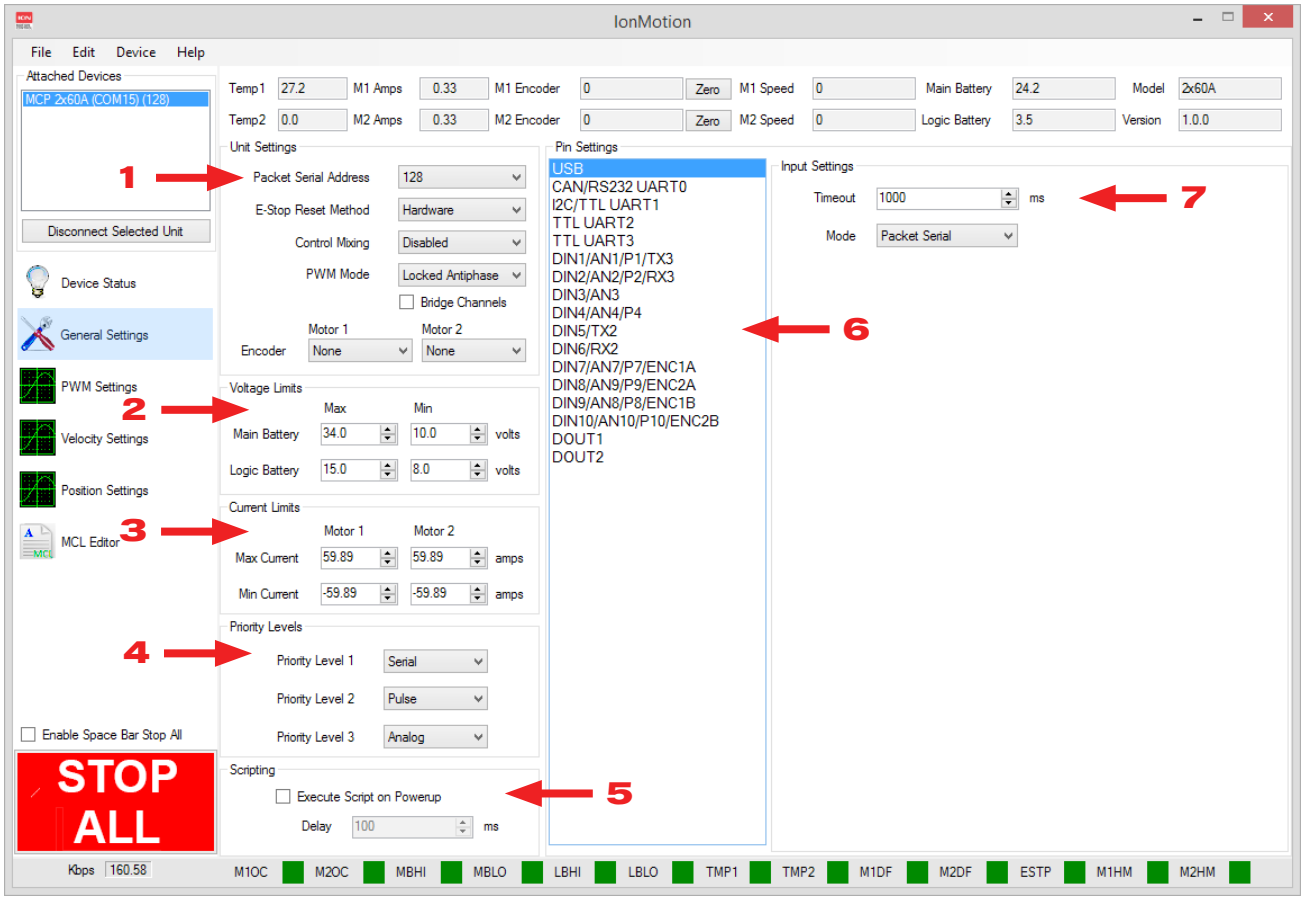
1.2.9 Help Menu



Label	Description
Roboclaw User Manual	Open Roboclaw User Manual in PDF reader.
MCP User Manual	Open MCP User Manual in PDF reader.
About Ion Motion	Version information for Ion Motion.

1.2.10 General Settings Screen

The general settings screen is used to configure the connected motor controller.



(1) Unit Settings

Function	Description
Packet Serial Address	Set the attached units address for bussed serial.
E-Stop Soft Reset	Select E-Stop Reset option. Defaults to hardware reset.
Control Mixing	Mixes two motor control channels for differential steering control.
PWM Mode	Locked Anti Phase or Sign Magnitude.
Bridge Channels	Bridge motor output channel 1 and channel 2.
Encoder input selection	Select the input signal pin for the motors encoder.

(2) Voltage Limits

Function	Description
Main Battery	Set minimum and maximum voltage cut off.
Logic Battery	Set minimum and maximum voltage cut off.

(3) Current Limits

Function	Description
Main Battery	Set minimum and maximum voltage cut off.
Logic Battery	Set minimum and maximum voltage cut off.

(4) Priority Levels

Function	Description
Priority Level 1	Sets input signal priorities. Level 1 is the master input. Level 2 inputs will override level 3. Used to time out a control input and move to a backup.
Priority Level 2	Sets input signal priorities. Level 1 is the master input. Level 2 inputs will override level 3. Used to time out a control input and move to a backup.
Priority Level 3	Sets input signal priorities. Level 1 is the master input. Level 2 inputs will override level 3. Used to time out a control input and move to a backup.

(5) Scripting

Function	Description
Execute Script on Power up	Enable to execute a MCL script on power up. It left uncheck any programmed script will not execute.
Delay	Sets run delay. Amount of time before scripts begins to execute after power up.

(6) Pin Settings

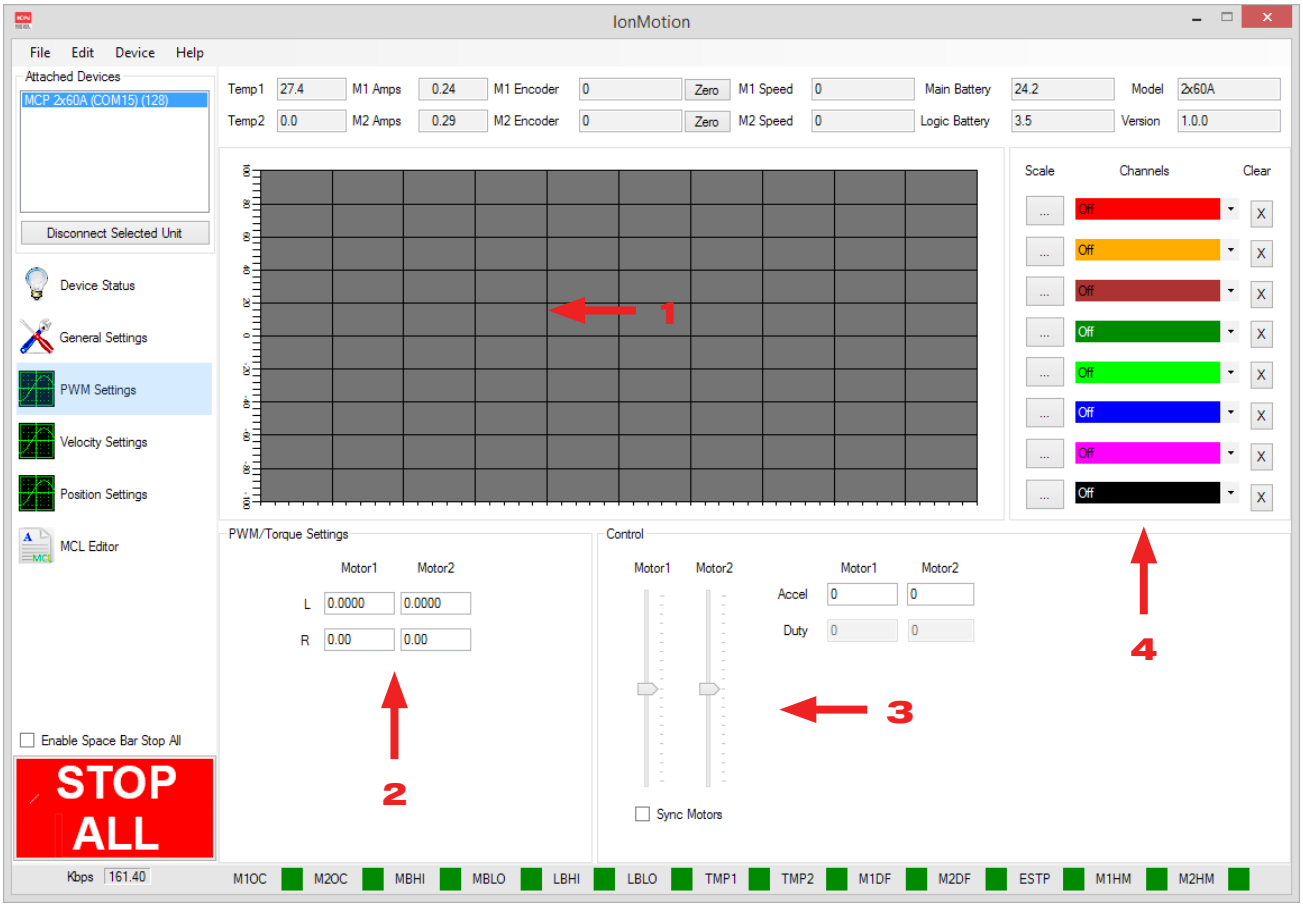
All available pin functions for a connected motor controller will be shown in the pin settings list. Each pin function shown in the list can have multiple settings. These settings will be displayed in the input settings (7)

(7) Input Settings

The input settings will change based on the highlighted pin in the pin setting list (6). Each pin setting is temporarily saved when changing to each pin. After all changes are made, the settings must be saved from the file menu.

1.2.11 PWM Settings

The PWM settings screen is used testing a connected motor controller. Sliders are provided to control each motor channel. This screen can also be used to determine the QPPS of attached encoders.



(1) Graph

Function	Description
Grid	Displays channel data with 100mS update rate and one second horizontal divisions.

(2) PWM/Torque Settings

Function	Description
L	MCP only. Motor Inductance in Henries.
R	MCP only. Motor resistance in Ohms.

(3) Control

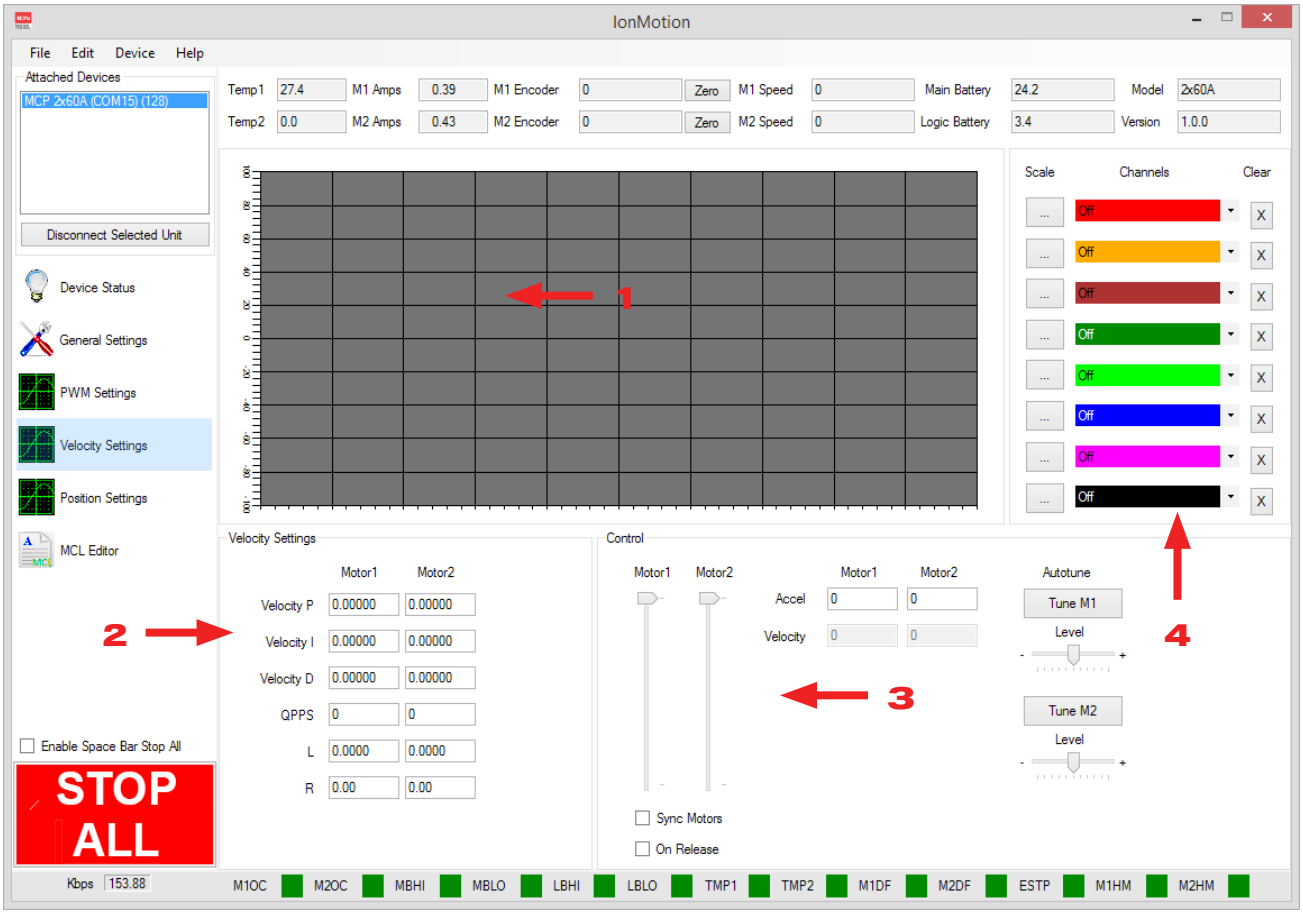
Function	Description
Motor 1	Controls motor 1 duty percentage forward and reverse.
Motor 2	Controls motor 2 duty percentage forward and reverse.
Sync Motors	Synchronises Motor 1 and Motor 2 Sliders.
Accel	Acceleration rate used when moving the sliders.
Duty	Displays the numberic value of the motor slider in 10ths of a Percent (0 to +/-1000).

(4) Graph Channels

Function	Description
Scale	Sets vertical scale to fit the range of the specified Channel.
Channels	Select data to display on the channel. The channel is graphed in the color shown. Channel options: • M1 or M2 Setpoint - User input for channel • M1 or M2 PWM - Motor PWM output • M1 or M2 Velocity - Motors Encoder Velocity • M1 or M2 Position - Motors Encoder Position • M1 or M2 Current - Motor running current • Temperature • Main Battery Voltage • Logic Battery Voltage
Clear	Clears channels graphed line.

1.2.12 Velocity Control Settings

The Velocity settings screen is used to set the encoder and PID settings for speed control. The screen can also used for testing and plotting.



(1) Graph

Function	Description
Grid	Displays channel data with 100mS update rate and one second horizontal divisions.

(2) Velocity Settings

Function	Description
Velocity P	Proportional setting for PID.
Velocity I	Integral setting for PID.
Velocity D	Differential setting for PID.
QPPS	Maximum speed of motor using encoder counts per second.
L	MCP only. Motor Inductance in Henries.
R	MCP only. Motor resistance in Ohms.

(3) Control

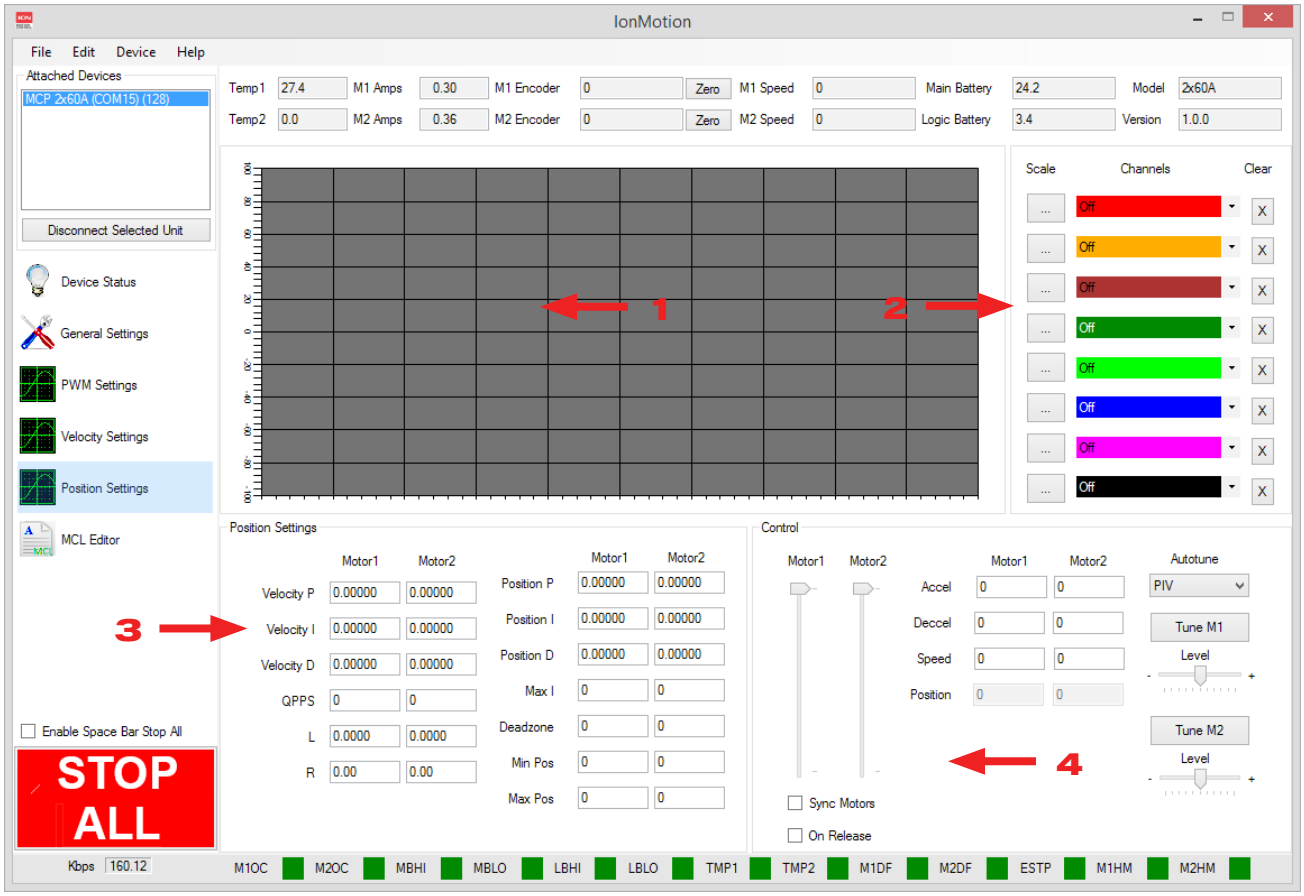
Function	Description
Motor 1	Motor 1 velocity control (0 to +/- maximum motor speed).
Motor 2	Motor 2 velocity control (0 to +/- maximum motor speed).
Sync Motors	Synchronises Motor 1 and Motor 2 Sliders.
On Release	Will not update new speed until the slider is released.
Accel	Acceleration rate used when moving the sliders.
Velocity	Shows the numeric value for the sliders current position.
Tune M1	Start motor 1 velocity auto tune.
Level	Adjust auto tune 1 values agressiveness. Sllide left for softer control.
Tune M2	Start motor 2 velocity auto tune.
Level	Adjust auto tune 2 values agressiveness. Sllide left for softer control.

(4) Graph Channels

Function	Description
Scale	Sets vertical scale to fit the range of the specified Channel.
Channels	Select data to display on the channel. The channel is graphed in the color shown. Channel options: • M1 or M2 Setpoint - User input for channel • M1 or M2 PWM - Motor PWM output • M1 or M2 Velocity - Motors Encoder Velocity • M1 or M2 Position - Motors Encoder Position • M1 or M2 Current - Motor running current • Temperature • Main Battery Voltage • Logic Battery Voltage
Clear	Clears channels graphed line.

1.2.13 Position Control Settings

The Position settings screen is used to set the encoder and PID settings for position control. The screen can also be used for testing and plotting.



(1) Graph

Function	Description
Grid	Displays channel data with 100mS update rate and one second horizontal divisions.

(2) Graph Channels

Function	Description
Scale	Sets vertical scale to fit the range of the specified Channel.
Channels	Select data to display on the channel. The channel is graphed in the color shown. Channel options: • M1 or M2 Setpoint - User input for channel • M1 or M2 PWM - Motor PWM output • M1 or M2 Velocity - Motors Encoder Velocity • M1 or M2 Position - Motors Encoder Position • M1 or M2 Current - Motor running current • Temperature • Main Battery Voltage • Logic Battery Voltage
Clear	Clears channels graphed line.

(3) Position Settings

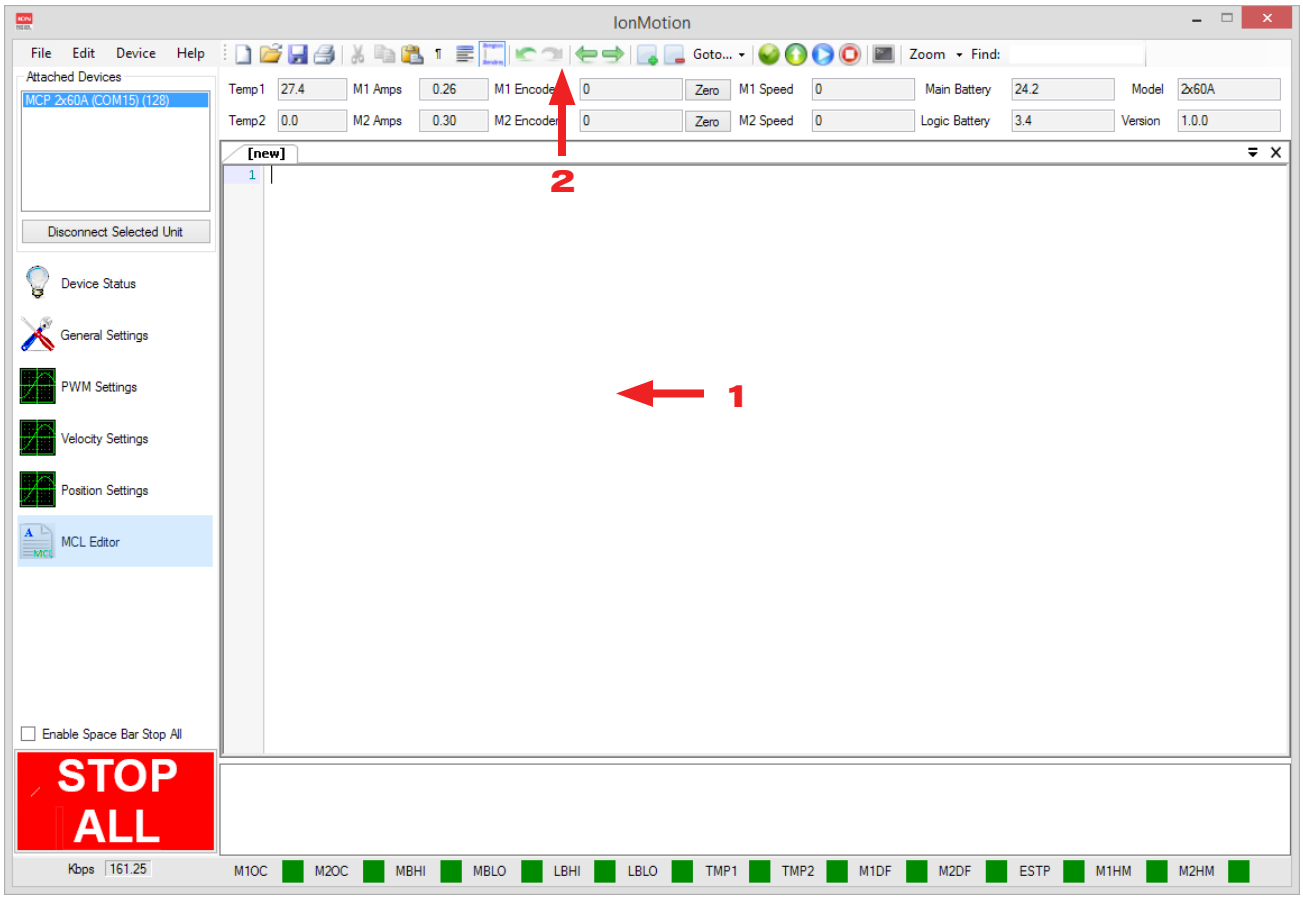
Function	Description
Velocity P	Proportional setting for velocity PID.
Velocity I	Integral setting for velocity PID.
Velocity D	Differential setting for velocity PID.
QPPS	Maximum speed of motor using encoder counts per second.
L	MCP only. Motor Inductance in Henries.
R	MCP only. Motor resistance in Ohms.
Position P	Proportional setting for position PID.
Position I	Integral setting for position PID.
Position D	Differential setting for position PID.
Max I	Maximum integral windup limit.
Deadzone	Zero position deadzone. Increases the "stopped" range.
Min Pos	Minimum encoder position.
Max Pos	Maximum encoder position.

(4) Control

Function	Description
Motor 1	Motor 1 velocity control (0 to +/- maximum motor speed).
Motor 2	Motor 2 velocity control (0 to +/- maximum motor speed).
Sync Motors	Synchronises Motor 1 and Motor 2 Sliders.
On Release	Will not update new speed until the slider is released.
Accel	Acceleration rate used when moving the sliders.
Deccel	Decceleration rate used when moving the sliders.
Speed	Speed to use with slide move.
Position	Numeric value of slider motor position.
Autotune	Method used. PD = Proportional and Differential. PID = Proportional Differential and Integral. PIV = Cascaded Velocity PD + Position P.
Tune M1	Start motor 1 velocity auto tune.
Level	Adjust auto tune 1 values agressiveness. Sllide left for softer control.
Tune M2	Start motor 2 velocity auto tune.
Level	Adjust auto tune 2 values agressiveness. Sllide left for softer control.

1.2.14 MCL Editor

The MCP editor is used to create MCL programs which are then download and ran on the MCP motor controller.



(1) MCL Editor

The built in program editor is a single document intereface. Mulitple files can be opened at a time and will be shown as tabs. The highlted tab is the working file.

(2) Control Toolbar

The editor toolbar controls all the fucntions for the MCL editor. Starting from left to right each function is listed in the following table.

Function	Description
New	Create new MCL file.
Open	Open existing MCL file.
Save	Save open MCL file.
Print	Print open MCL file.
Cut	Standard cut text function.
Copy	Standard copy text function.

Function	Description
Paste	Standard paste text function.
Shown Hidden Characters	Shows hidden characters that are typically left over from a copy / paste function.
Highlight Current Line	Highlights current selected line.
Show Folding Lines	Start motor 1 velocity auto tune.
Undo	Undo changes.
Redo	Redo changes.
Add	
Remove	
Goto Bookmarks	
Check	Check current script for syntax errors.
Upload	Uploaded current program to attached motor controller.
Run Script	Execute script on attached motor controller.
Stop Script	Stop current running script on attached motor controller.
Open Terminal Window	Open standard terminal window for communications to attached motor controller.
Set Text Zoom	Set text zoom size.
Find	Search current program.

1.3 Firmware Updates

1.3.1 Ion Studio Setup

Download and install the Ion Studio application. Win7 or newer is required. When opening Ion Studio, it will check for updates and search for a USB Windows Driver to verify installation. If the USB driver is not found, Ion Studio will install it.

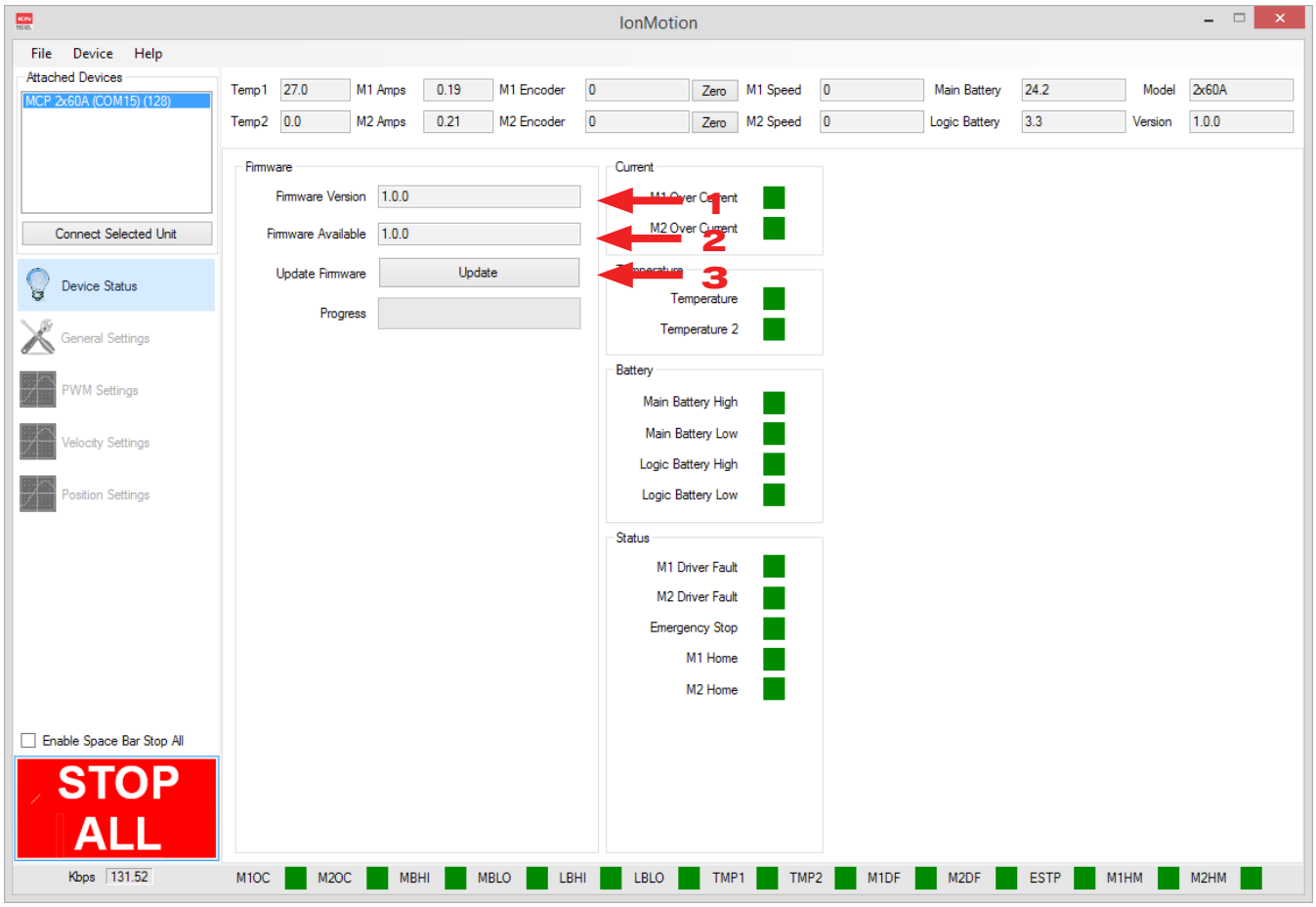
1. Open the Ion Studio application.

2. Apply a reliable power source such as a fully charge battery to power the motor controller.

3. Connect the powered motor controller to a USB port on your computer with Ion Studio already open.

1.3.2 Firmware Update

Once Ion Studio detects the motor controller it will display the current firmware version in the Firmware Version field (1). Each time Ion Studio is started it will check for a new version of its self which will always include new firmware. If an update is required Ion Studio will download the latest version and display it in the firmware available field (2).



1. When a new version of firmware is shown click the update button (3) to start the process.

2. Ion Studio will begin to update the firmware. While the firmware update is in progress the onboard LEDs will begin to flash. The onboard flash memory will first be erased. It is important power is not lost during this process or the motor controller will no longer function. There is no recovery if power fails during the erase process.

3. Once the firmware update is complete the motor controller will reset. Click the "Connect Selected Unit" button to re-connect.

1.4 DC Power Settings

1.4.1 Automatic Battery Detection on Startup

Auto detect will sample the main battery voltage on power up or after a reset. All Lipo batteries, depending on cell count will have a minimum and maximum safe voltage range. The attached battery must be within this acceptable voltage range to be correctly detected. Undercharged or overcharged batteries will cause false readings and the MCP will not properly protect the battery. If the automatic battery detection mode is enabled using the on-board buttons, the Stat2 LED will blink to indicate the battery cell count that was detected. Each blink indicates the number of LIPO cells detected. When automatic battery detection is used the number of cells detected should be confirmed on power up.

Undercharged or overcharged batteries can cause an incorrect auto detection voltage.

1.4.2 Manual Voltage Settings

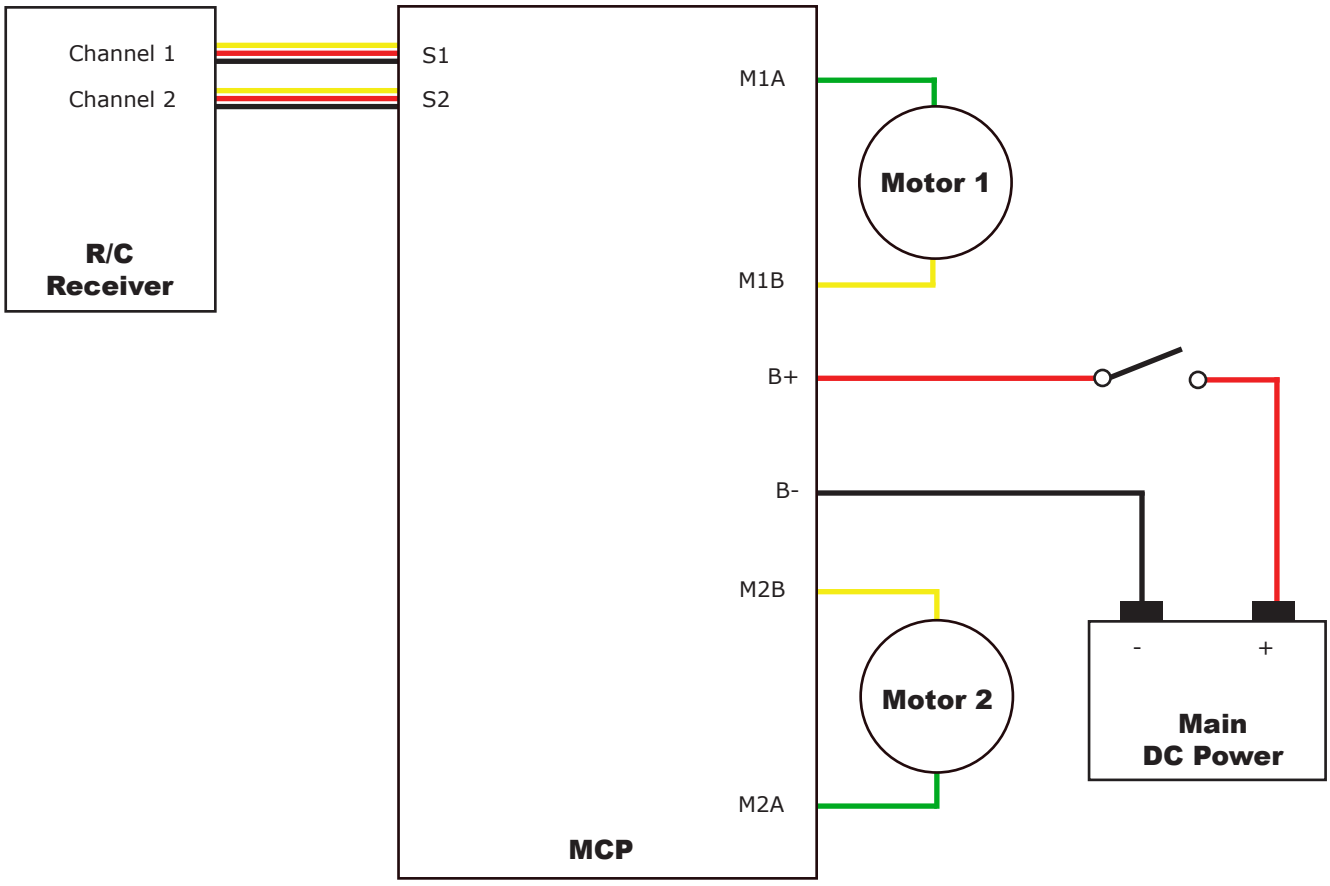
The minimum and maximum voltage can be set using the Ion Studio application or packet serial commands. Values can be set to any value between the boards minimum and maximum voltage limits. This feature can be useful when using a power supply to power the MCP. A minimum voltage just below the power supply voltage of 2VDC will prevent the power supply voltage from dipping too low under heavy load. A maximum voltage set to just above the power supply voltage 2VDC will help protect the power supply from regenerative voltage spikes if an external voltage clamp circuit is not being used. However when the minimum or maximum voltages are reached MCP will go into either braking or freewheel mode. This feature will only help to protect a power supply not correct regenerative voltages issues. A voltage clamping circuit is required to correct any regenerative voltage issues when a power supply is used as the main power source. See Voltage Clamping.

1.5 Wiring

1.5.1 Basic Wiring

The MCP has many control modes and each mode may have unique wiring requirements to ensure safe and reliable operation. The diagram below illustrates a very basic wiring configuration used in a small motor system where safety concerns are minimal. This is the most basic wiring configuration possible. Any wiring of MCP controllers should include a main battery shut off switch, even when safety concerns are minimal. Never underestimate a motorized system in an uncontrolled condition.

In addition, the MCP is a regenerative motor controller. If the motors are moved when the system is off, it can cause potential erratic behavior due to the regenerative voltages powering the system. A return path to the battery should always be supplied if the system can move when main power is disconnected or a fuse is blown by adding a power diode across the shut off switch.



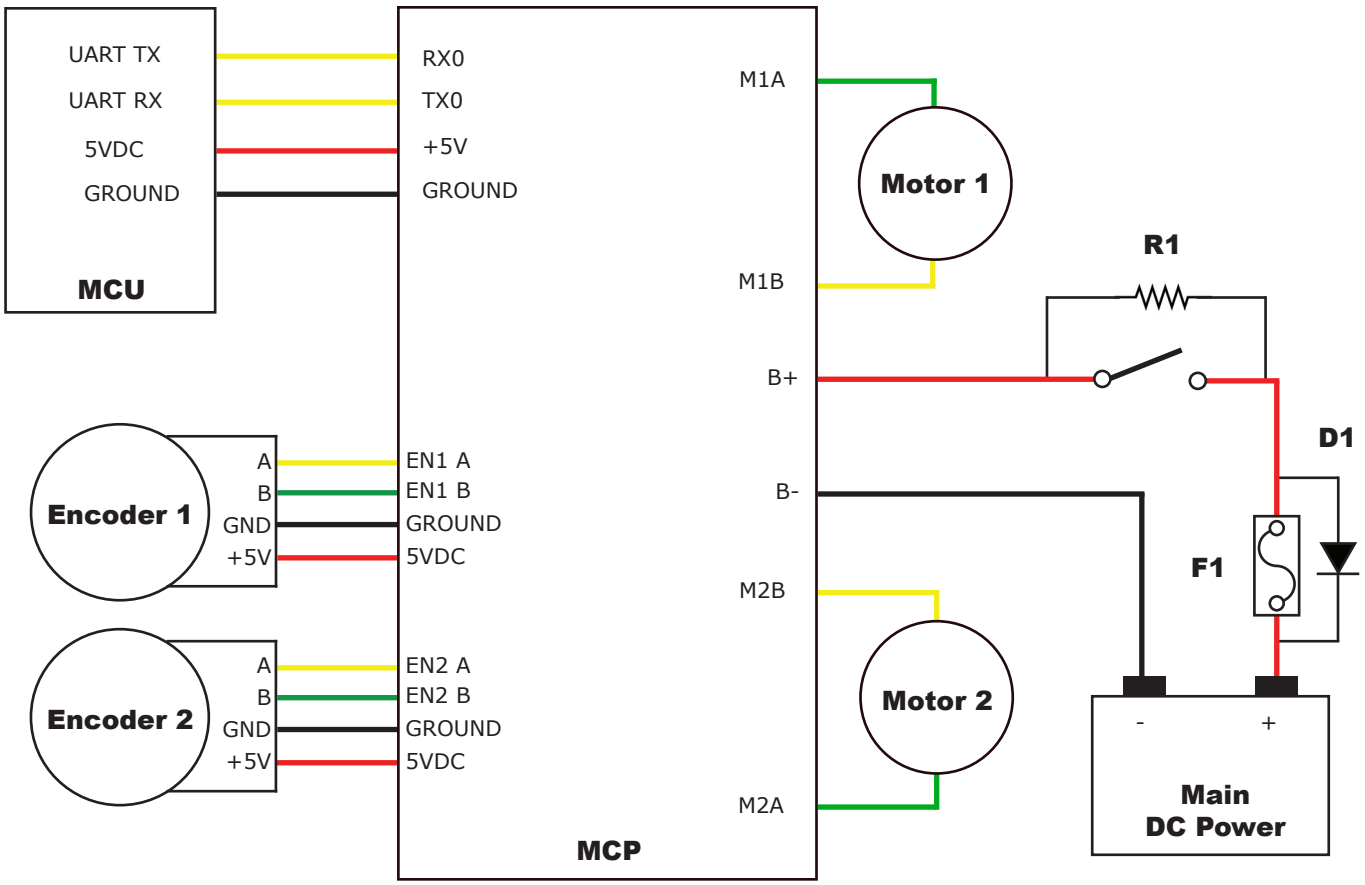
Never disconnect the negative battery lead before disconnecting the positive!

1.5.2 Wiring Safety

In all system with movement, safety is a concern. The wiring diagram below illustrates a properly wired system with several safety features. An external main power cut off is required for safety. When the RoboClaw is switched off or the fuse is blown, a high current diode (D1) is required to create a return path to the battery for any regenerative voltages. The use of a pre-charge resistor (R1) is required to avoid high inrush currents and arcing when using high voltages. A pre-charge resistor (R1) could be 220ohm, 1/2Watt for a MCP266 60VDC motor controller with a pre-charge time of about 200 milliseconds and an in-rush current of less than 1 amp.

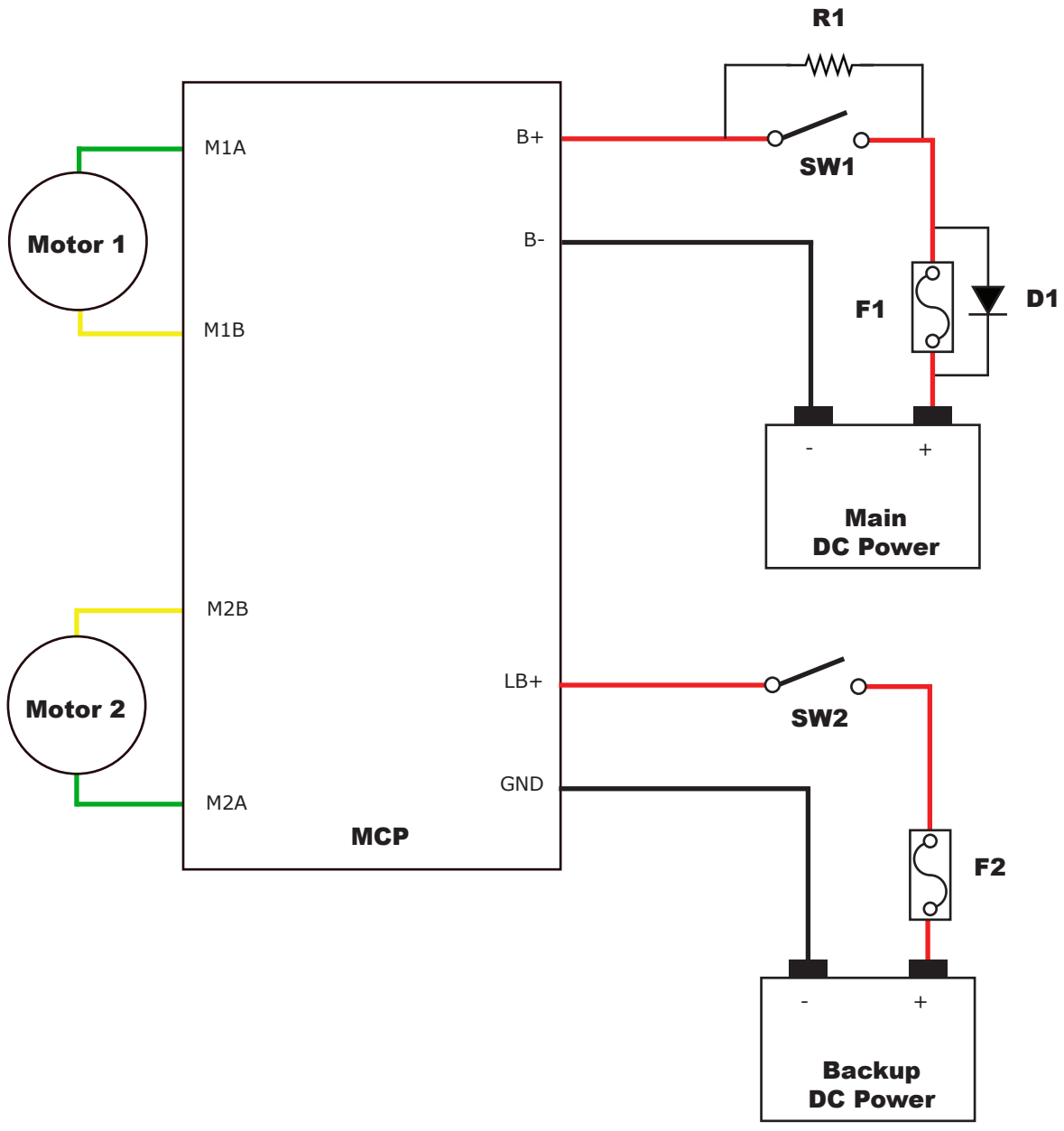
1.5.3 Wiring Closed Loop Mode

A wide range of inputs are supported including quadrature encoders, pulse width encoders, analog absolute encoders and hall effect sensors for closed loop operation. See the Encoder section of this manual for additional information.



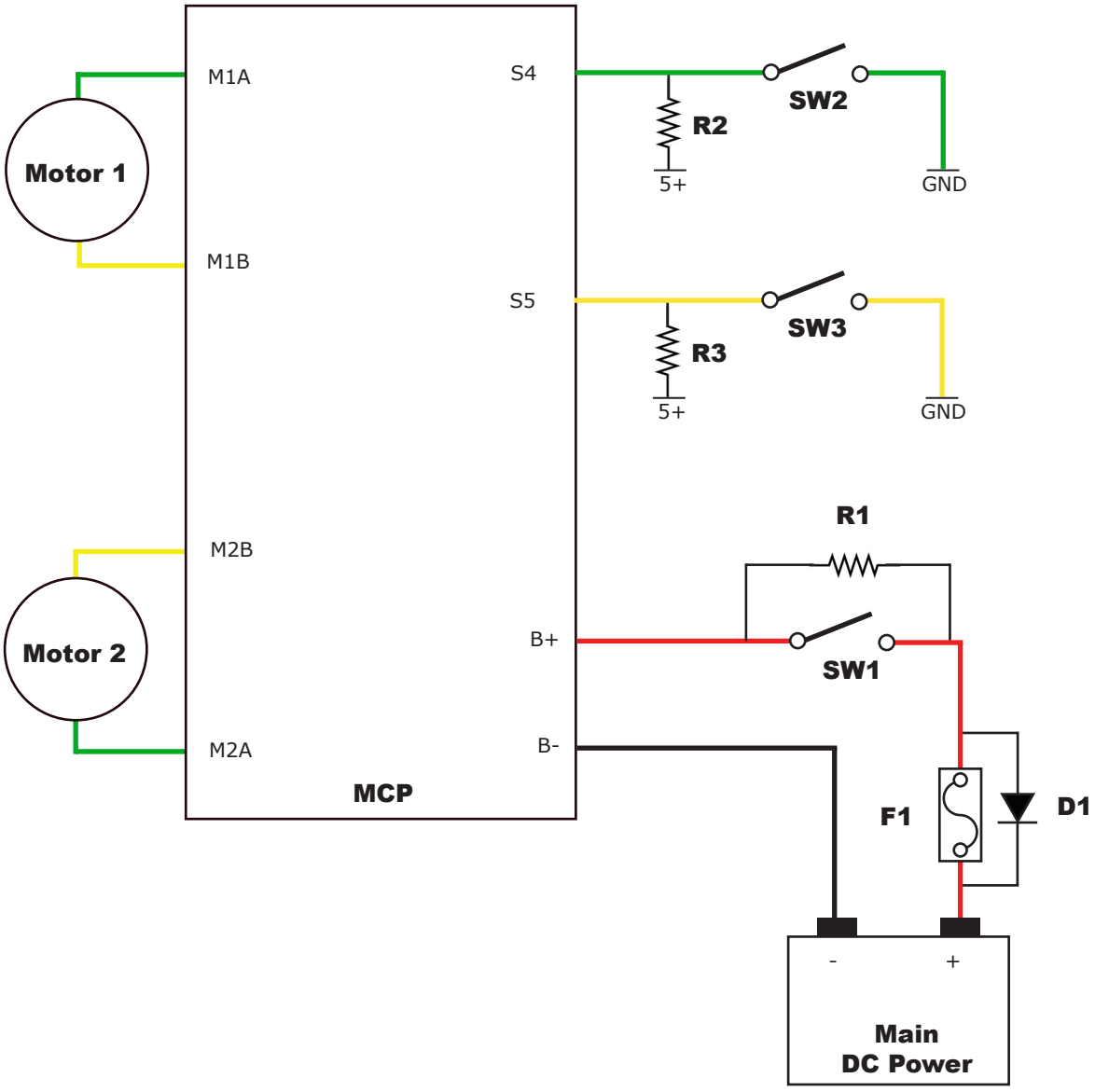
1.5.4 Backup Power

An optional backup battery is supported. Under heavy loads the main power can suffer voltage drops, causing potential logic brown outs if the voltage dips below approximately 8VDC, which may result in uncontrolled behavior. A separate power source for the motor controllers logic circuits, can remedy potential problems from main power voltage drops. The backup battery maximum input voltage is 15VDC with a minimum input voltage of 6VDC. The 5VDC regulated user output is supplied by the secondary backup battery if supplied. The mAh of the backup battery should be determined based on the load of attached devices powered by the regulated 5VDC user output. The MCP controller will automatically use the appropriate power source depending on the current voltage of either power source.



1.5.5 Limit, Home and E-Stop Wiring

Digital or Analog inputs can be used for Limit, Home or E-Stop signalling. A pull-up resistor to 5VDC or 3.3VDC should be used as shown. The circuit below shows a NO (normally open) style switch. Connect the NO to the specific input pin and the COM end to a ground shared with the MCP.

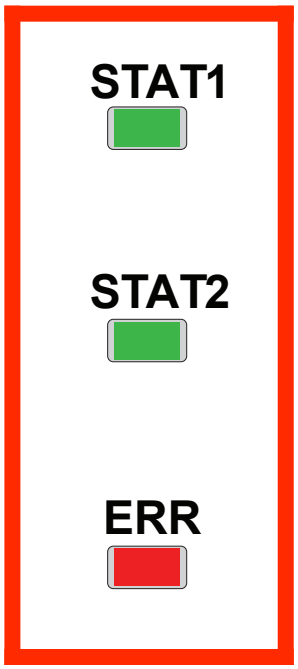


1.6 Status LEDs

1.6.1 Status and Error LEDs

MCP includes 3 LEDs to indicate status. Two green status LEDs labeled STAT1 and STAT2 and one red error LED labeled ERR. When the motor controller is first powered on all 3 LEDs should blink briefly to indicate all LEDs are functional.

The LEDs will behave differently depending on the mode. During normal operation the status 1 LED will remain on continuously or blink when data is received in RC Mode or Serial Modes. The status 2 LED will light when either drive stage is active.



1.6.2 Message Types

There are 3 types of message RoboClaw can indicate. The first type is a fault. When a fault occurs, both motor channel outputs will be disabled and RoboClaw will stop any further actions until the unit is reset, or in the case of non-latching E-Stops, the fault state is cleared. The second message type is a warning. When a warnings occurs both motor channel outputs will be controlled automatically depending on the warning condition. As an example if an over temperature of 85c is reach RoboClaw will reduce the maximum allowed current until a safe temperature is reached. The final message type is a notice. Currently there is only one notice indicated.

1.6.3 LED Blink Sequences

When a warning or fault occurs RoboClaw will use the LEDs to blink a sequence. The below table details each sequence and the cause.

LED Status	Condition	Type	Description
All three LEDs lit.	E-Stop	Error	Motors are stopped by braking.
Error LED lit while condition is active.	Over 85c Temperature	Warning	Motor current limit is recalculated based on temperature.
Error LED blinks once with short delay. Other LEDs off.	Over 100c Temperature	Error	Motors freewheel while condition exist.
Error LED lit while condition is active.	Over Current	Warning	Motor power is automatically limited.
Error LED blinking three times.	Logic Battery High	Error	Motors freewheel until reset.
Error LED blinking four times.	Logic Battery Low	Error	Motors freewheel until reset.
Error LED blinking five times.	Main Battery High	Error	Motors are stopped by braking until reset.
Error LED lit while condition is active.	Main Battery High	Warning	Motors are stopped by braking while condition exist.
Error LED lit while condition is active.	Main Battery Low	Warning	Motors freewheel while condition exist.
STAT1 and STAT2 LEDs cycling back and forth after power up.	MCP is waiting for new firmware.	Notice	MCP is in boot mode. Use Ion Studio setup utility to update firmware which will clear the notice.

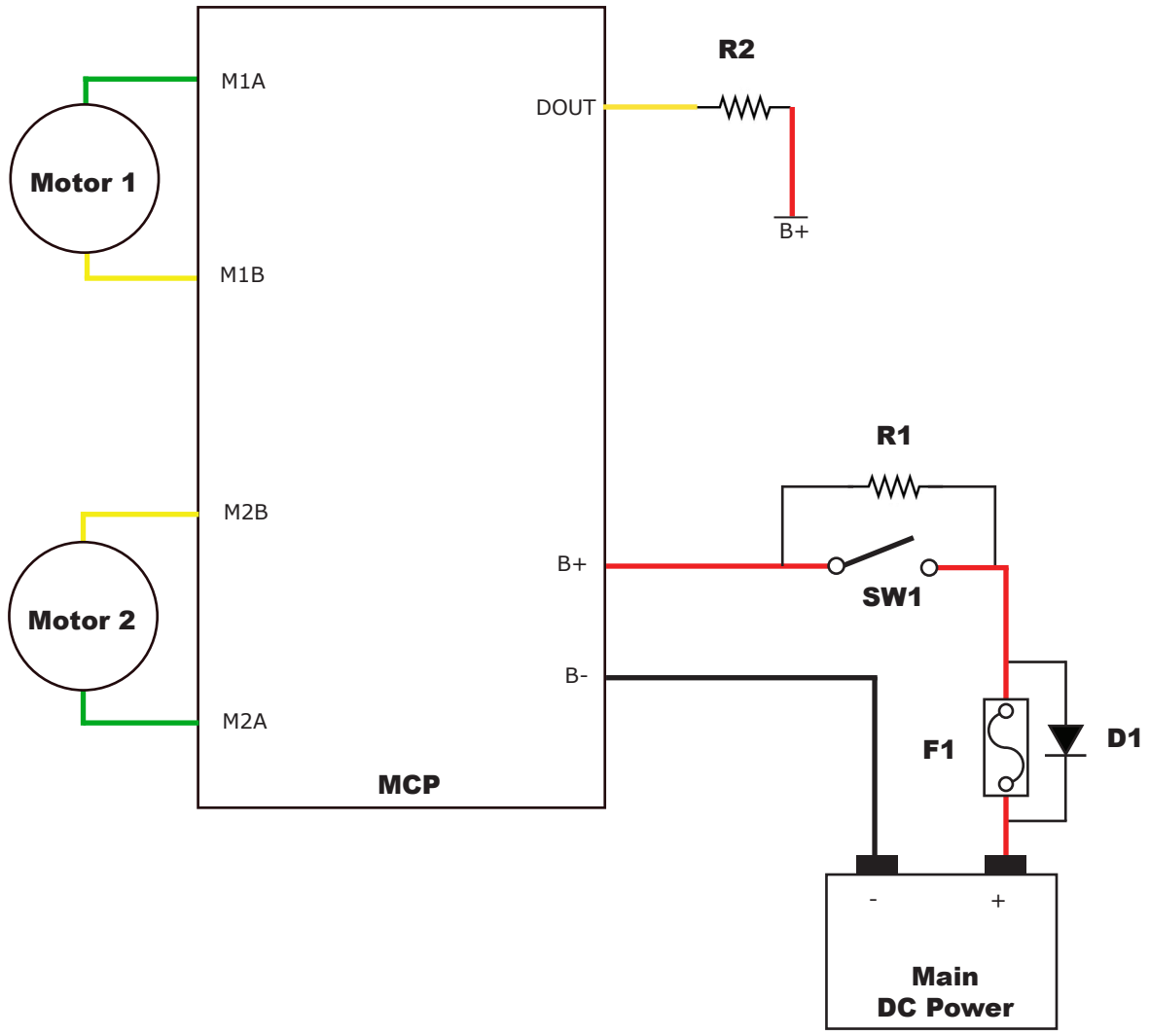
1.7 Regenerative Voltage Clamping

1.7.1 Voltage Clamp

When using power supplies regenerative voltage spikes will need to be dissipated. A power resistor in the range of 10 Ohms at 50 watts for small motors and down to 1 Ohms or lower for larger motors may be required. 50 watts will likely cover most situations however smaller wattage resistor may also work depending on the application. The regenerative energy will be dissipated as heat.

1.7.2 Simple Voltage Clamp Setup

When using DOUT pins as a direct voltage clamp, the total power dissipated cannot exceed the 3A at 40VDC limit of the DOUT pins. The DOUT pins can be double up to increase the current capacity. The maximum rate voltage of 40VDC will not change. To use the DOUT pins as a direct voltage clamp set the specific DOUT pin action to "Over Voltage" to activate on an Over Voltage using Ion Studio.

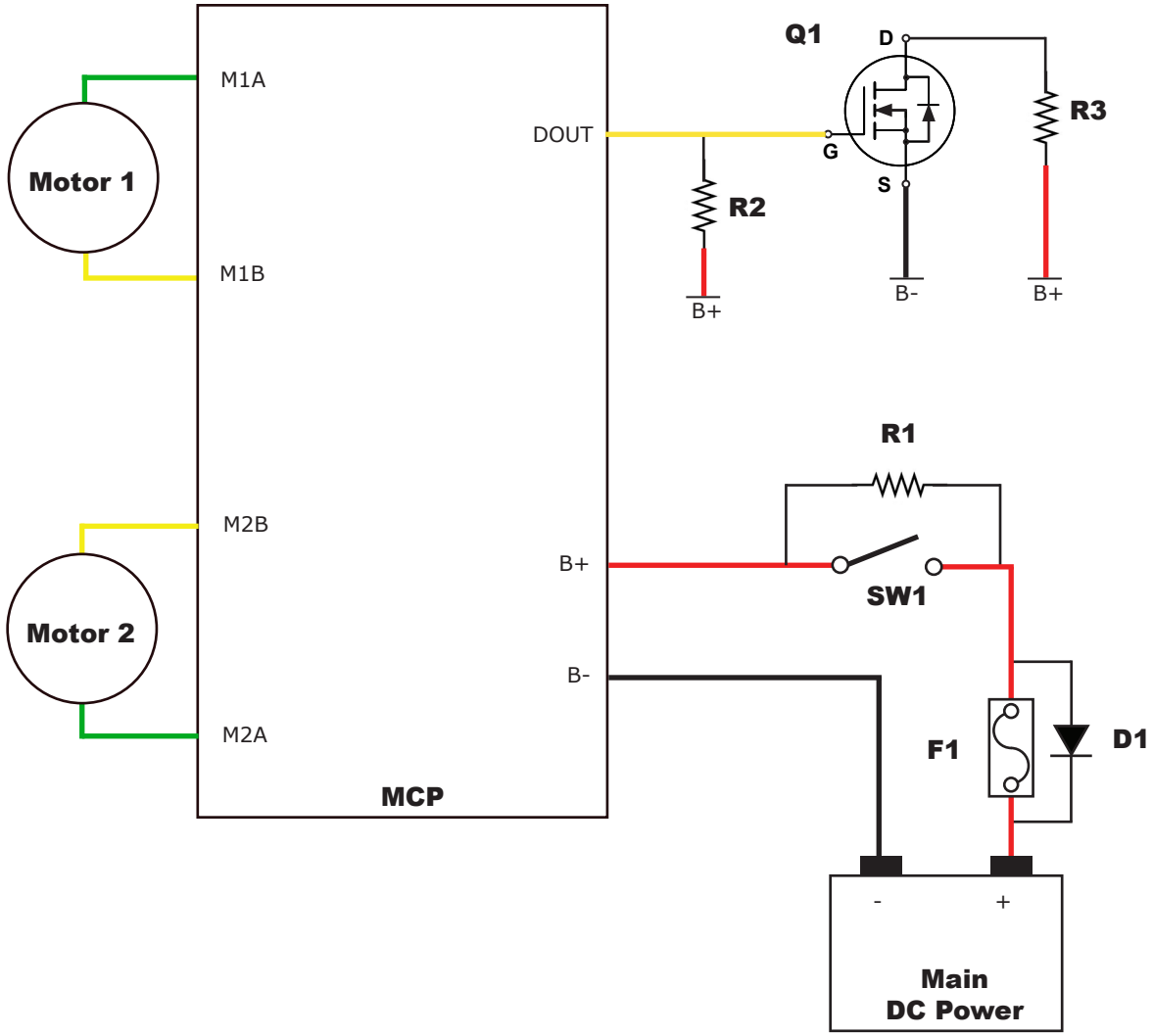


1.7.3 MOSFET Voltage Clamp Setup

When using DOUT to drive an N-Channel MOSFET set the specific DOUT pin action to "Over Voltage" and check the "inverted" option in Ion Studio.

1.7.4 MOSFET Voltage Clamp Wiring

Alternatively a solid state switch (Q1) can be used along with the power resistor(R3) for higher current requirements. Wire the circuit as shown below. Q1 should be a 5v logic level N-Channel MOSFET. An example would be the FQP30N06L. It is rated to a maximum voltage of 60VDC. You may need to change the MOSFET used based on the application. The gate of the MOSFET must be connected to a DOUT pin along with a 5VDC pull-up resistor.



1.7.5 Voltage Clamp Setup and Testing

The circuits shown will need to be modified for each application to make sure it properly dissipates the regenerative voltages. Testing should consist of running the motor up to 25% of its speed and then quickly slowing down without braking or E-Stop while checking the voltage spike generated. Repeat, increasing the speed and power by 5% each time and checking the voltage spike again. Repeat this process until 100% power is achieved or excessive over voltage is detected. If over voltages are not clamped within requirements, either a lower Ohm resistor will be required or additional DC Link capacitance will need to be added across B+ and B-. Adding additional DC Link capacitance increases the amount of time the power resistor will have to dissipate the regenerated power.

1.8 Bridged Channel Mode

1.8.1 Bridging Channels

The dual channels on an MCP can be bridged to run as one channel, effectively doubling its current capability for one motor. The MCP can be damaged if it is not set to bridged channel mode before wiring up the single motor as show. Download and install Ion Studio setup utility. Connect the motor controller to the computer using an available USB port. Run Ion Studio and in general settings check the option to bridge channels. Then click "Save Settings" in the device menu. When operating in bridged channel mode the total peak current output is combined from both channels. The peak current run time is dependant on heat build up. Adequate cooling must be maintained.

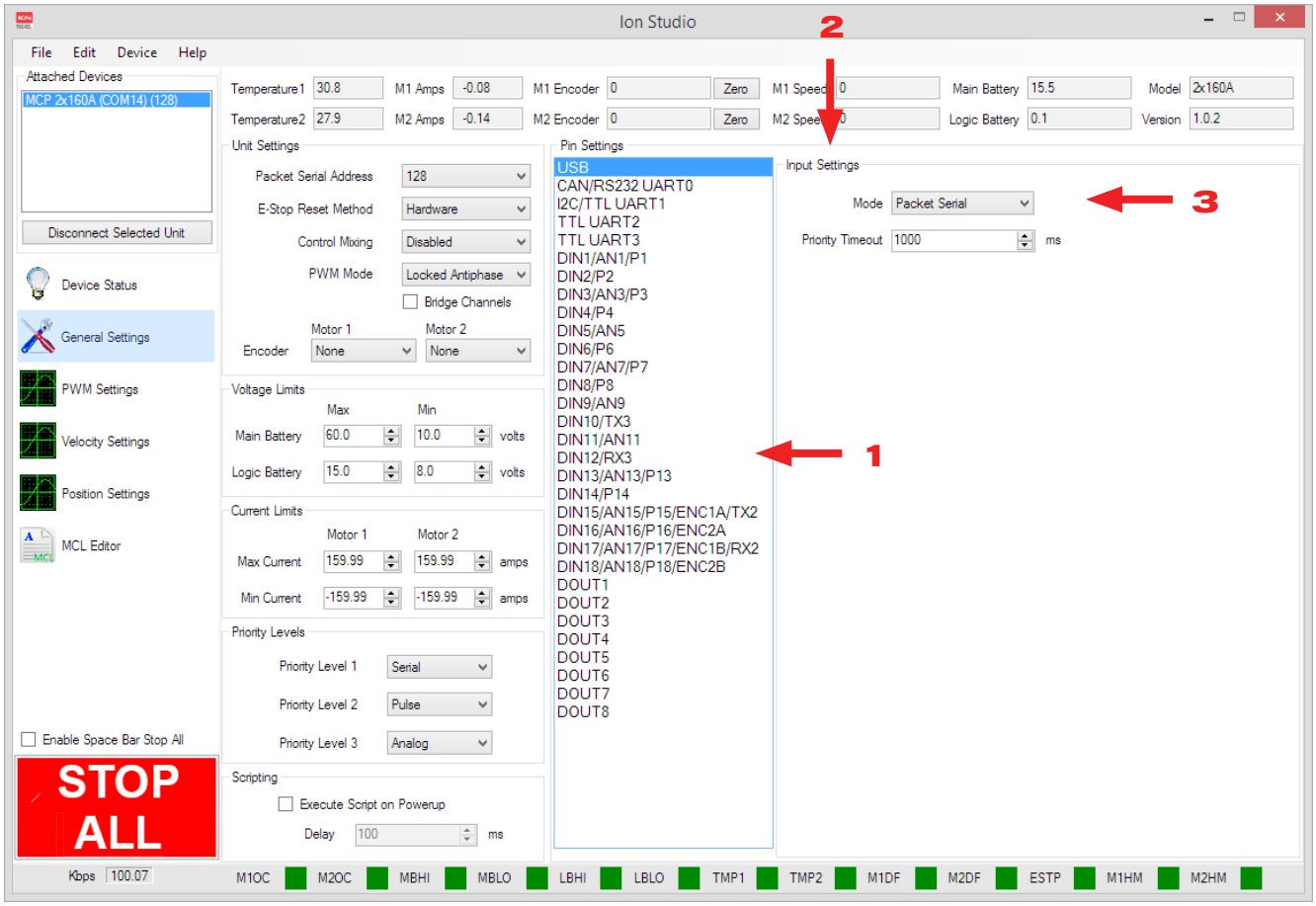
1.8.2 Bridged Channel Wiring

When bridged channel mode is active the internal driver scheme for the output stage is modified. The output leads must be wired correctly or damage will result.

1.9 I/O Configuration

1.9.1 I/O Types

There are several I/O types available on the MCP. This includes Digital Input, Pulse Inputs, Analog, CAN, TTL Serial, RS-232 and USB. The I/O types can be set using Ion Studio. Under the general settings screen all available pin settings will be listed for the connected motor controller. Refer to the data sheet for pin locations of the specific model of motor controller.



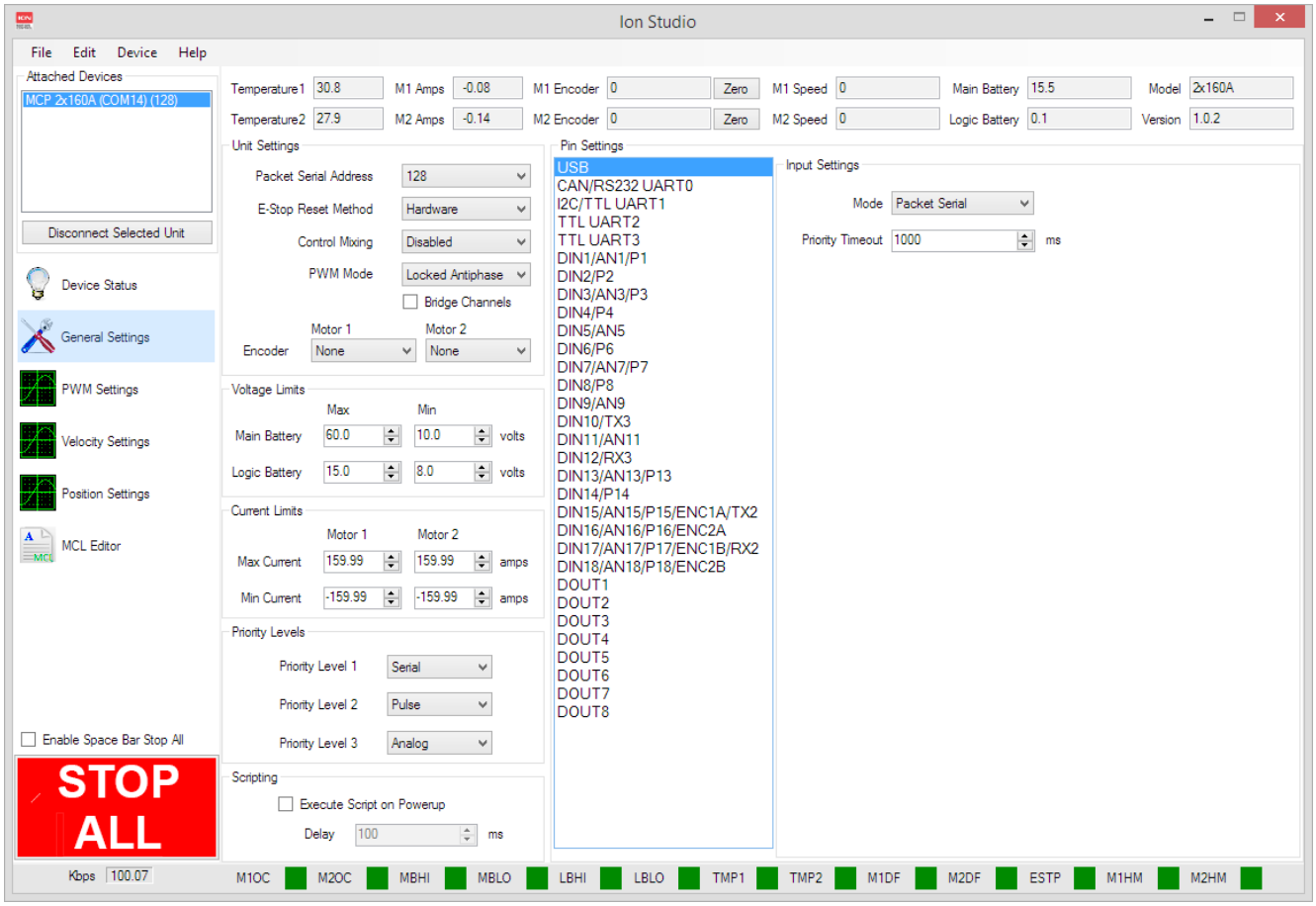
1.9.2 Pin Settings

Pins can have more than one function. When a pin is selected from the pin settings list (1) the available functions for that pin will be displayed under the mode drop down (3) shown under input settings (2). The mode drop down menu (3) enables the desired function for the selected pin. Once the mode is set the available configuration options can be adjusted. Some configuration options may not be applicable to the selected mode and will be grayed out.

Once all changes are made the settings can be saved to the connected MCP by choosing "Menu-> Save Settings". In addition to read the settings of a connected MCP choose "Menu-> Read Settings".

1.9.3 USB

The USB has two modes and a timeout function. The USB can be access from an MCL program with a priority timeout. To configure the USB control settings select USB from the pin settings list.



Modes

Pins can have multiple functions. The mode drop down menu under input settings is used to set the main fuction of the selected pin from the pin settings list. After the mode is set the available settings for that mode will be active and can be set. If the setting is not applicable to the selected mode it will be grayed out.

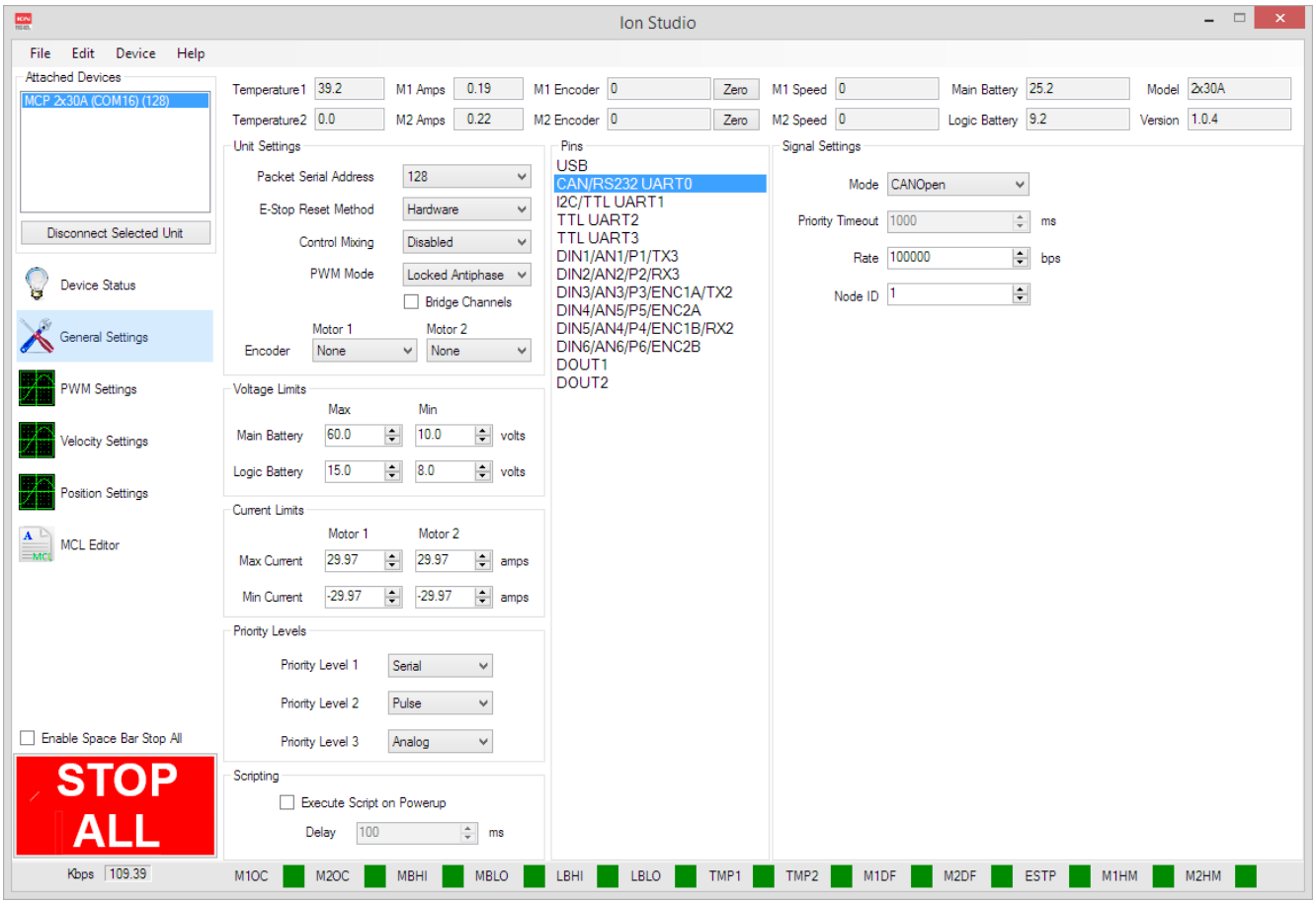
- **Packet Serial** - Used to communicate with the MCP firmware. If set to different mode Packet Serial can be forced active by setting the RTS signal of the Virtual Serial port.

- **MCL Serial** - MCL Serial is used to give an MCL script access to USB. Can be used with the CDC ACM virtual comport over the USB connection.

  Priority Timeout - Time in milliseconds new data must be received before timing out. As long as any data is received, even if it is not a valid packet the timeout period will be reset.

1.9.4 CANOpen

To configure the CANopen control settings select the CAN pins from the pin settings list. Then select CANopen from the mode drop down menu to enable the CAN buss feature.



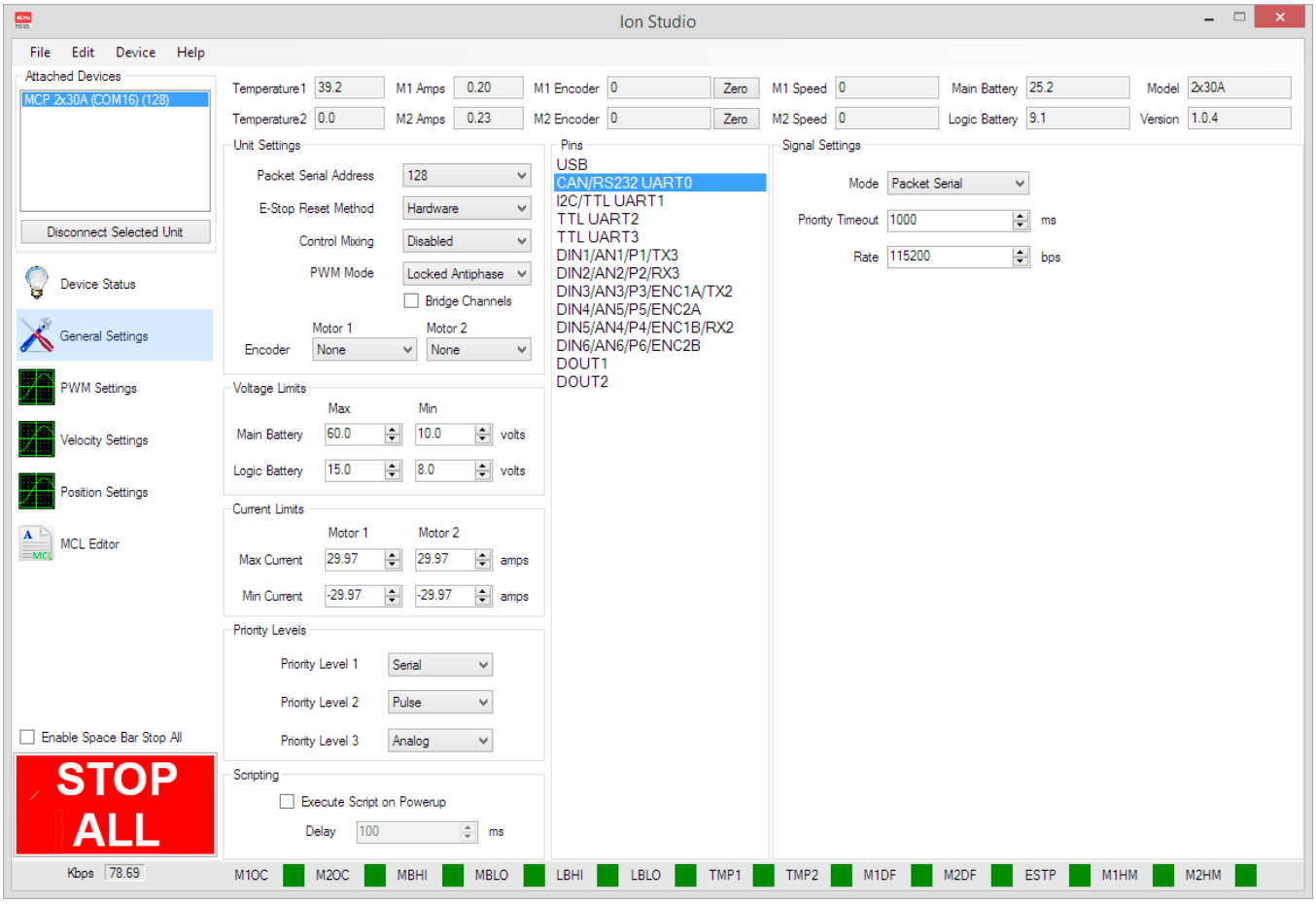
Priority Timeout - Disabled, CANOpen functions will execute at all priority levels.

Rate - Bit rate for CANopen. Show in bits per secopnd. Default value is 250000. Supported bit rates are: 1000000, 800000, 500000, 250000, 125000, 100000, 50000, 20000 and 10000

Node ID - Sets teh Node ID for the CANOpen Interface (valid values 1 to 127).

1.9.5 RS-232

The RS-232 port is hardware level converted. It can be directly interfaced with any standard RS-232 equipment. CANopen pins are typically shared and when not in use are set to a high Z state. To configure the RS-232 control settings select RS-232 from the pin settings list.



Modes

Pins can have multiple functions. The mode drop down menu under input settings is used to set the main fuction of the selected pin from the pin settings list. After the mode is set the available settings for that mode will be active and can be set. If the setting is not applicable to the selected mode it will be grayed out.

- **Packet Serial** - Mode to communicate to the MCP firmware. Packet Serial mode can be forced active by setting the RTS signal of the Virtual Serial port.

  Priority Timeout - Time in milliseconds new data must be received before timing out. As long as any data is received, even if it is not a valid packet the timeout period will be reset.

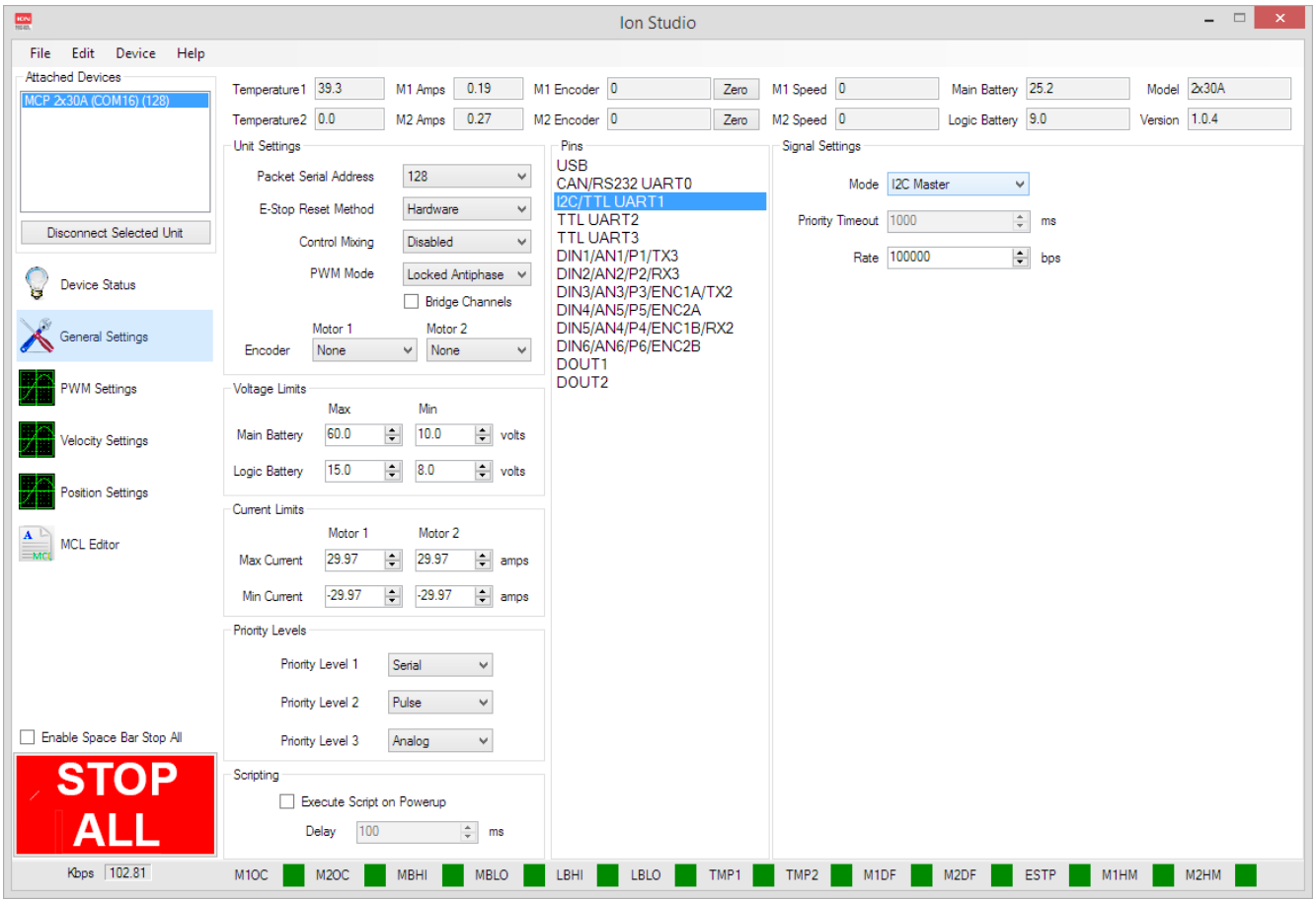
  Rate - The bit rate to communicate at. Default value is 115200. Valid Range is 300 to 1000000 bps.

- **MCL Serial** - Give an MCL script access to the UART. Can be used with the CDC ACM virtual comport over the USB connection.

  Rate - The bit rate to communicate at. Default value is 115200. Valid Range is 300 to 1000000 bps.

1.9.6 I2C

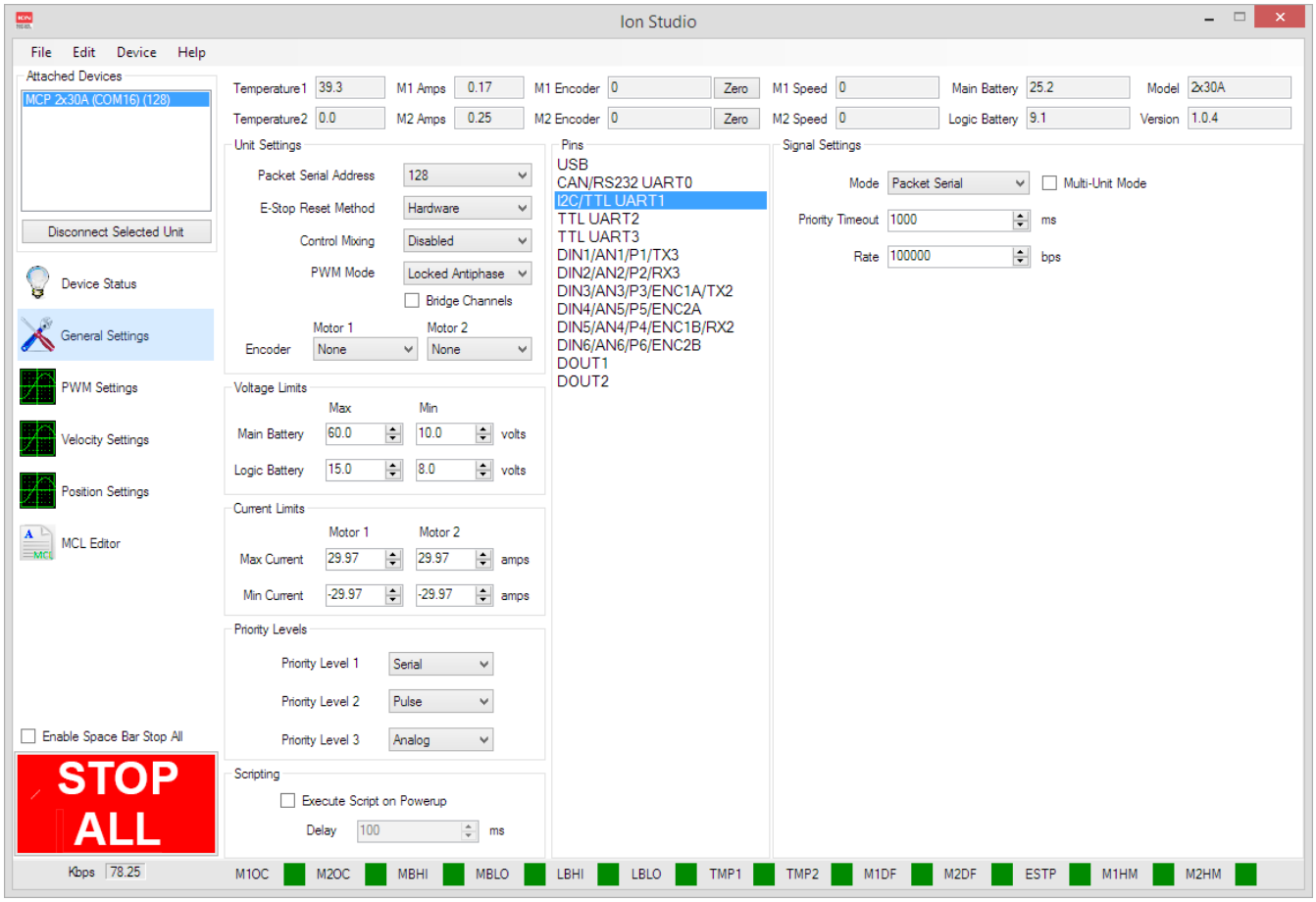
Enable the I2C bus control settings. MCP supports one I2C master bus which can be used for controlling sensors and memory I2C slave devices. I2C is only accessable from a user MCL script.



Rate - I2C Bit Rate used to communicated to slave devices. Valid bit rates for I2C Master is 100000, 400000 and 1000000.

1.9.7 TTL UARTx

Configure UARTx control settings. TTL UARTs are buffered. There are a total of three TTL UARTS available.



Modes

Pins can have multiple functions. The mode drop down menu under input settings is used to set the main fuction of the selected pin from the pin settings list. After the mode is set the available settings for that mode will be active and can be set. If the setting is not applicable to the selected mode it will be grayed out.

- **Packet Serial** - Mode to communicate to the MCP firmware. Packet Serial mode can be forced active by setting the RTS signal of the Virtual Serial port.

  Priority Timeout - Time in milliseconds new data must be received before timing out. As long as any data is received, even if it is not a valid packet the timeout period will be reset.

  Rate - The bit rate to communicate at. Default value is 115200. Valid Range is 300 to 1000000 bps.

  Multi Unit Mode - Sets TTL output into open-drain mode, allowing multiple MCP units to use the same RX connection to the external controller. An external pullup resistor (recommend values 1k to 10k) should be added to the external controllers RX pin.

- **MCL Serial** - Give an MCL script access to the UART. Can be used with the CDC ACM virtual comport over the USB connection.

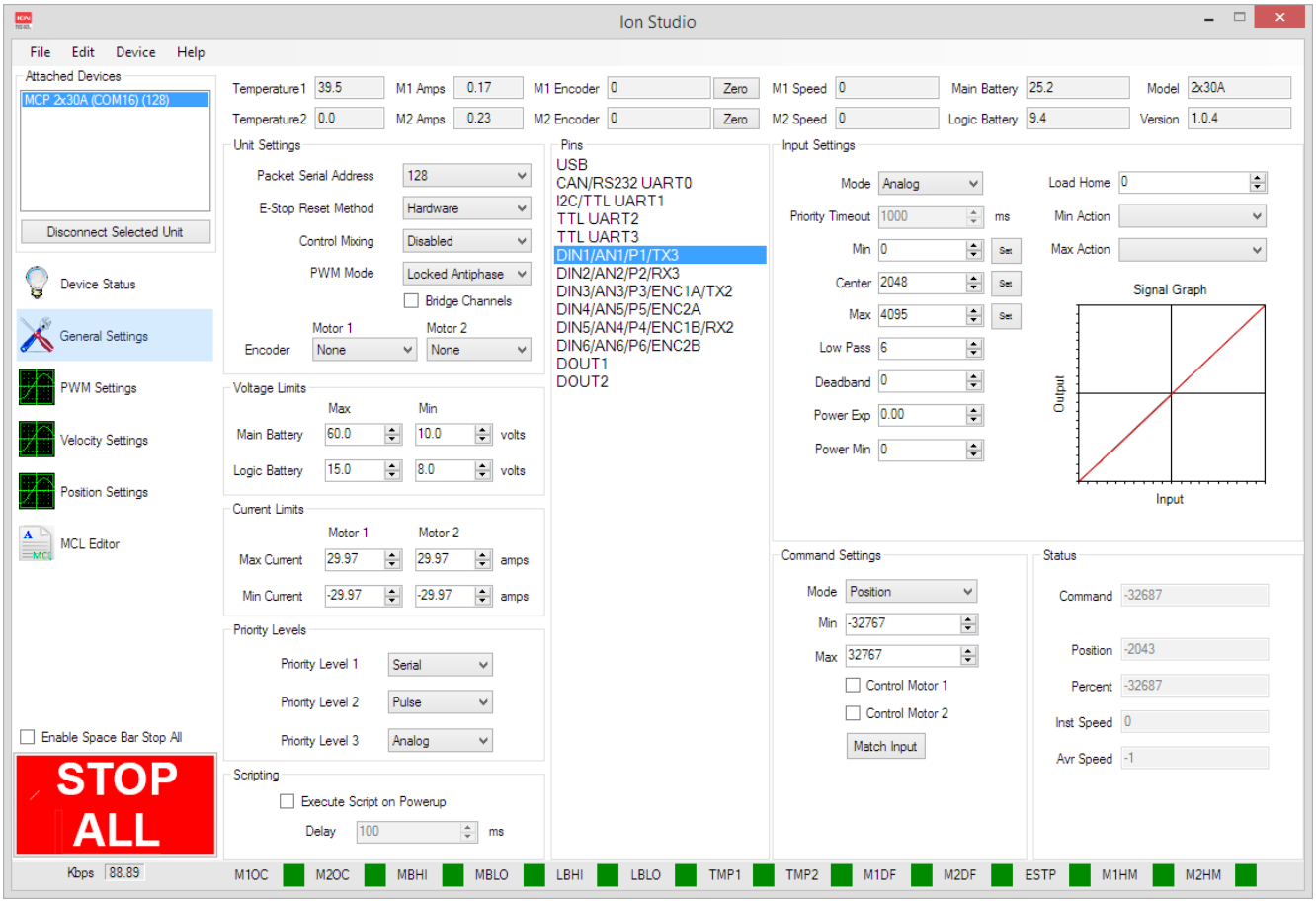
  Rate - The bit rate to communicate at. Default value is 115200. Valid Range is 300 to 1000000 bps.

  Multi Unit Mode - Sets TTL output into open-drain mode, allowing multiple MCP units to use the same RX connection to the external controller. An external pullup resistor (recommend values 1k to 10k) should be added to the external controllers RX pin.

- **Disabled Mode** - The MCP units have multiple functions on most I/O pins. To use the DIN functions on UART2 and or UART3 pins that are multiplexed need to be disabled. The DIN pins for UART2 and 3 are marked with TX2/RX2 and TX3/RX3 respectively. If the UART is not set tpo disabled, the multiplexed DIN pins functions will be disabled.

1.9.8 Digital, Analog, Pulsed and Encoder Inputs

Pins with a DINx can be configured as digital inputs. Pins with ANx can be configure as analog inputs. Pins with a Px can be configred as pulse inputs. Pins with an ENCx are can be configured as encoder inputs. The encoder inputs must be grouped. ENC1A and ENC1B are used for encoder one which corresponds to motor channel one. Not all pins are capable of all three functions. The pin settings list will display the connected motor controllers available pins and fuctions.



Input Settings

All configuration options shown in the Input Settings section apply to Digital, Analog and Pulse inputs. The settings will reset each time the mode is changed. To read the settings of a connected MCP choose "Menu-> Read Settings". Once all changes are made the settings can be saved to the connected MCP by choosing "Menu-> Save Settings".

Priority Timeout - Only applies to pulsed input pins. The timeout period is set in milliseconds. The selected input will wait for a signal or pin state change for the specified amount of time. If no signal or state change is detected the next priority level pin will be selected. This system is only active if a motor is controlled by the input signal.

Min, Center and Max - The Min, Center and Max values are used to calculate the Position value of an input signal. Unlike Analog inputs, Digital inputs only have 2 inherant values, 0 and 4095. However by using the digital low pass filter, a PWM input signal on the Digital pin can be converted into a value from 0 to 4095. The resolution and update rate will depend on the PWM frequency and the low pass filter sampling rate.

Low Pass - Sets the low pass digital filter for the input. The digital filter takes a running average of 2^N samples at 10khz. By default the low pass filter is disabled on digital inputs. To use the digital input with a PWM signal (without using pulse measurement) the low pass filter must be enabled. Note this method of reading a PWM signal supports PWM frequecnies up to 5khz (half the sample rate). Frequencies higher than that should either use analog inputs with an external filter or pulse width measurement. To determine the best sample size divide 10000 by the update rate of the input signal provided. If udpated readings of 20 times a second are required, 500 samples per reading will be the maximum. The closest 2^N value less than or equal to 200 is 256. The low pass setting would be 8.

Deadband - Specify the input signal range around center that will be considered equal to 0. For example if an analog joystick is attached to the input, the spring that holds the joystick centered may not always return back to exactly 0. Deadband allows you to set a window that will be considered 0.

Power Exp - Add an exponential response curve to the input signal. Positive values increase sensitivity close to the center point while negative values increase input sensitivity near the Min and Max limits.

Power Min - Adds a +/- offset to the input signal around the center setting. Input values greater than or less than zero (as calculated using the deadband) will have the Power Min offset added / subtracted from the input. For example, if a motor needs a minimum of 5% duty to start moving an offset using Power Min can be added to always start the motor properly and increase the input range. If a value of 5 is choosen the duty for the motor at stop will be 0 and when movement is intiallized will jump to 5% duty to start the motors movement.

Load Home - The value that is set to the controlled motors encoder register if a load home action is triggered by the selected input.

Min Action - Drop down list of automated actions that can be performed if the input signal is less than or equal to the set Min value from Min, Center and Max.

Max Action - Drop down list of automated actions that can be performed if the input signal is less than or equal to the set Min value from Min, Center and Max.

Min Actions and Max Actions

The Min and Max actions are preset automated functions that can be perform if the input values of the selected pin are equal to or less than the value set for Min and greater than the value set for Max.

M1/M2 Safe Stop - When triggered will stop the controlled motor at the maximum decceleration rate supported. The motor will remain stopped until the input signal comes back to Center

M1/M2 Off - The controlled motor will turn off (coast to a stop). Motor will be prevented from new motion until the action is cleared.

M1/M2 Reverse - The controlled motors direction will be reversed(at the currently set accel / deccel rate)

M1/M2 Forward Limit - The controlled motor will be stopped at the maximum decceleration rate allowed. After stopped the motor will be prevented from moving forward, however reverse commands will be allowed until the action is cleared.

M1/M2 Backward Limit - The controlled motor will be stopped at the maximum decceleration rate allowed. After stopped the motor will be prevented from moving backward, however forward commands will be allowed until the action is cleared.

M1/M2 Load Home - Load a specified value to motors encoder register. Allows reset of non 0 values.

Run Script - Start running the programmed user script.

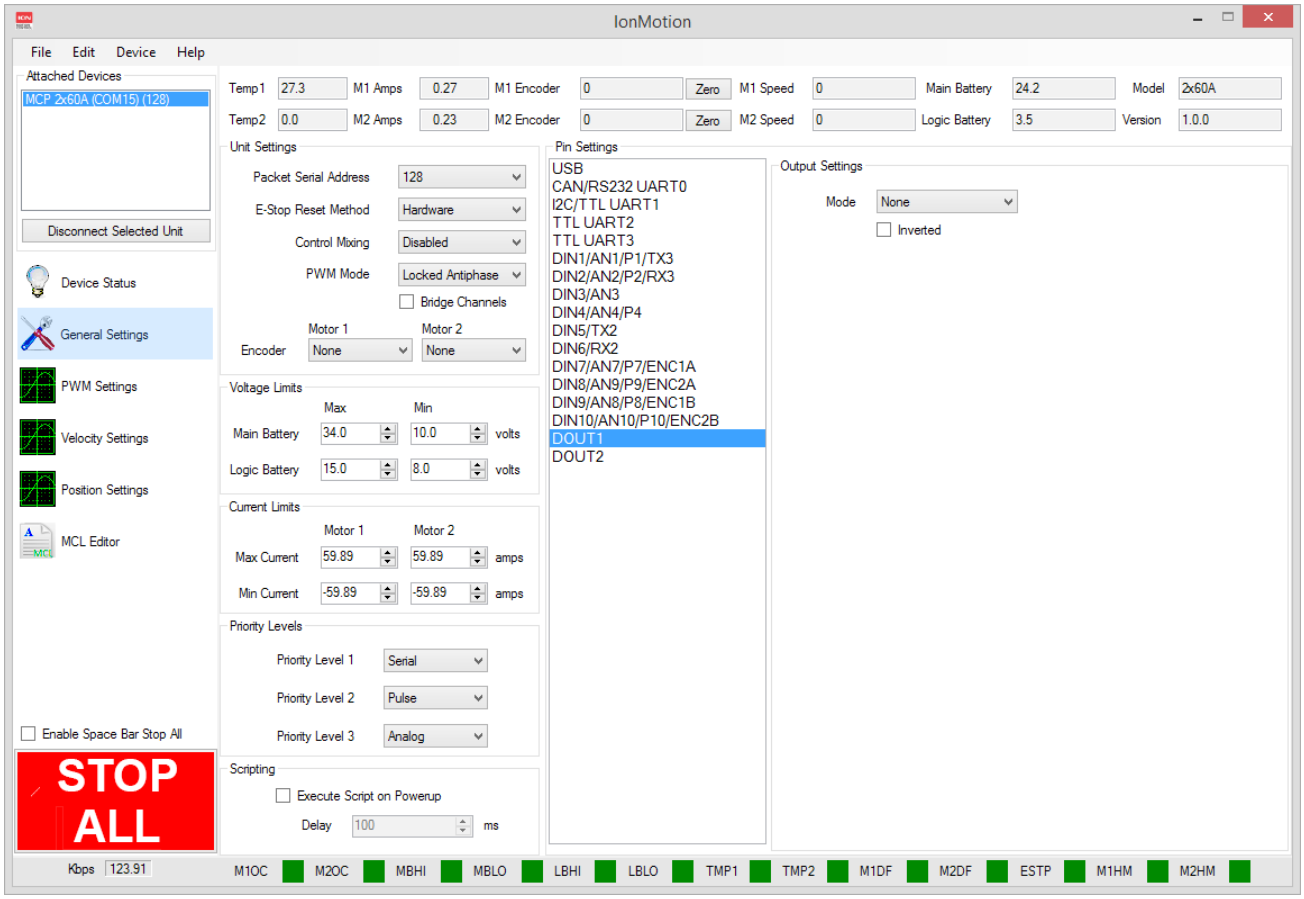
Stop Script - Stop user script

Reset Script - Reset user script to begining, keep running if already running.

E-Stop - Stop both motors at maximum supported deceleration rate.

1.9.9 DOUTx

The DOUTx pins are sinking only. They can be used to drive loads such as contactors or braking systems. Configure the MCP digital output pins.



Mode

Sets the action that will trigger the output pin. MCL scripts can override the set action

Motor1 On - Turns the selected DOUTx pin on when motor channel 1 turns on.

Motor2 On - Turns the selected DOUTx pin on when motor channel 2 turns on.

Motors On - Turns the selected DOUTx pin on when either motor channel turns on.

Motor 1 Reverse - Turns the selected DOUTx pin on when motor channel 1 reverses direction.

Motor 1 Reverse - Turns the selected DOUTx pin on when motor channel 1 reverses direction.

Motors Reverse - Turns the selected DOUTx pin on when either motor channel reverses direction.

Over Voltage - Turns the selected DOUTx pin on when the motor controller has an over voltage condition.

Over Temperature - Turns the selected DOUTx pin on when the motor controller has an over temperature condition.

Status1 LED - Turns the selected DOUTx pin on when the led goes active. Can be used to add remotely located duplicate LEDs.

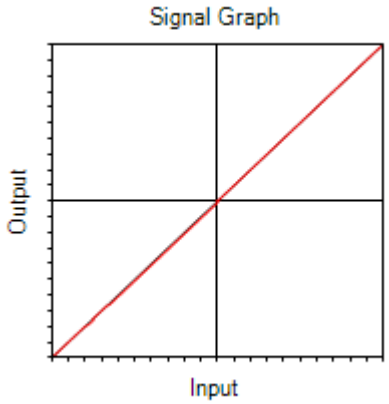
Status2 LED - Turns the selected DOUTx pin on when the led goes active. Can be used to add remotely located duplicate LEDs.

Error LED - Turns the selected DOUTx pin on when the led goes active. Can be used to add remotely located duplicate LEDs.

Invert
The DOUTx is normally off (open). The inverted setting will turn the DOUTx pin on (closed) on power up and off (open) when active.

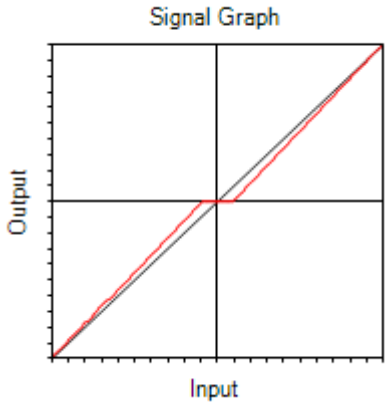
1.9.10 Signal Graph

The signal graph shows how the input signal is modified by the input settings. If no settings are changed and the Min, Center and Max ranges are set to their defaults a straight slope will be shown. The graph below shows the default signal slope.

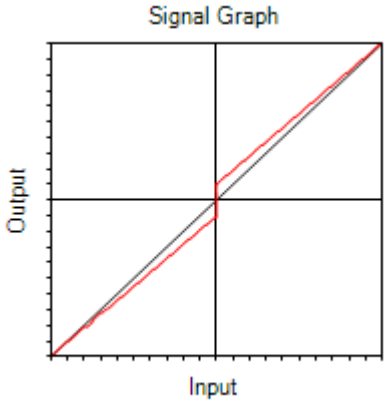


Deadband

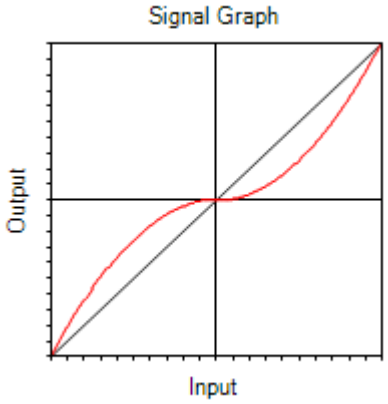
If an R/C signal is used to control a motor and deadband is added so the R/C controller stick doesn't need to be exactly centered to stop the controlled motor. The graph would show an horizontal in the middle of the graph at the 0 point. The lines would slop up and down an angle like normal after the shown zero area. The size of the horizontal line at the 0 point will changes it length based on how much deadband is set.



Power Minimum
In another example, some motors may need an offset in the minimum power to start rotating. After adjusting the Power Min to the required minimum value to start rotating the graph would show a vertical offset of the sloped lines above and below the center / zero point.

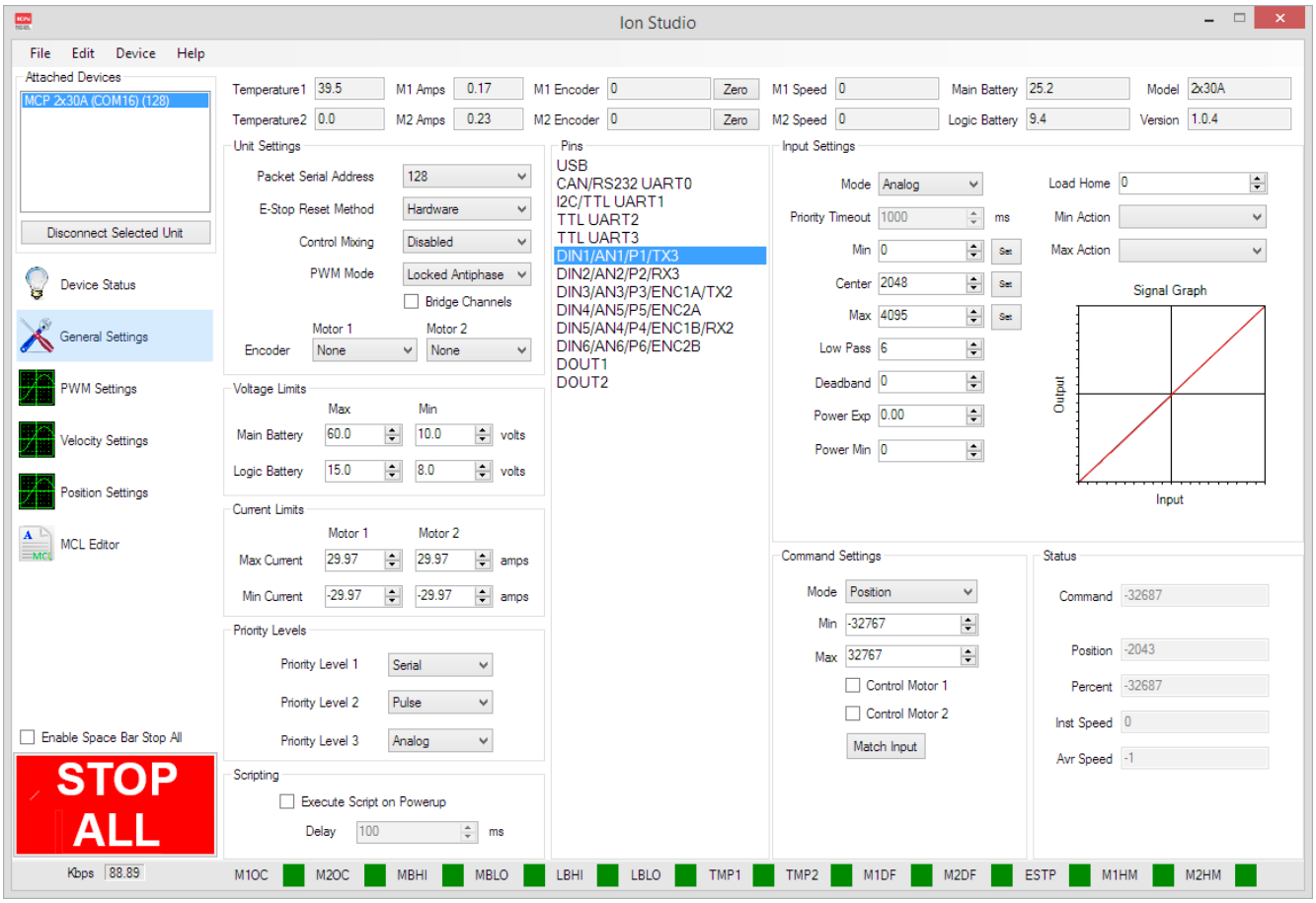


Power Exponential
In the last example, when the Power Exp is adjusted the graph will show a curve in or out giving more resolution at the lower input values or at the higher input values depending on whether the Power Exp setting is positive or negative.



1.9.11 Command Settings

The signal graph shows how the input signal is modified by the input settings. If no settings are changed and the Min, Center and Max ranges range and set to their defaults a straight slope will be shown.



Modes

Inputs can be used to control one or both motors by duty cycle, velocity control or position control depending on the PID settings for that particular motor. Each input has its base value (Position), however inputs also have other inherant states. The current percentage of the set range, the instantaneous speed and the average speed. The command used to control the selected motor(s) can be any of these values. For example when using a jog dial with a quadrature encoder input the instant or average speed of the input signal (the Jog dial turn speed) can be used to control the speed of the motor(s).

Min and Max

The Min and Max command range sets the scale of the motor control singal. For example the default settings of +/- 32,767 scale the input signal so its full range will output the full range of the motors duty cycle, velocity or position ranges. If the application requires the motor not run at 100% power or full speed the scale can be reduced, individually, in the forward in the reverse directions by reducing the Command range Min and Max.

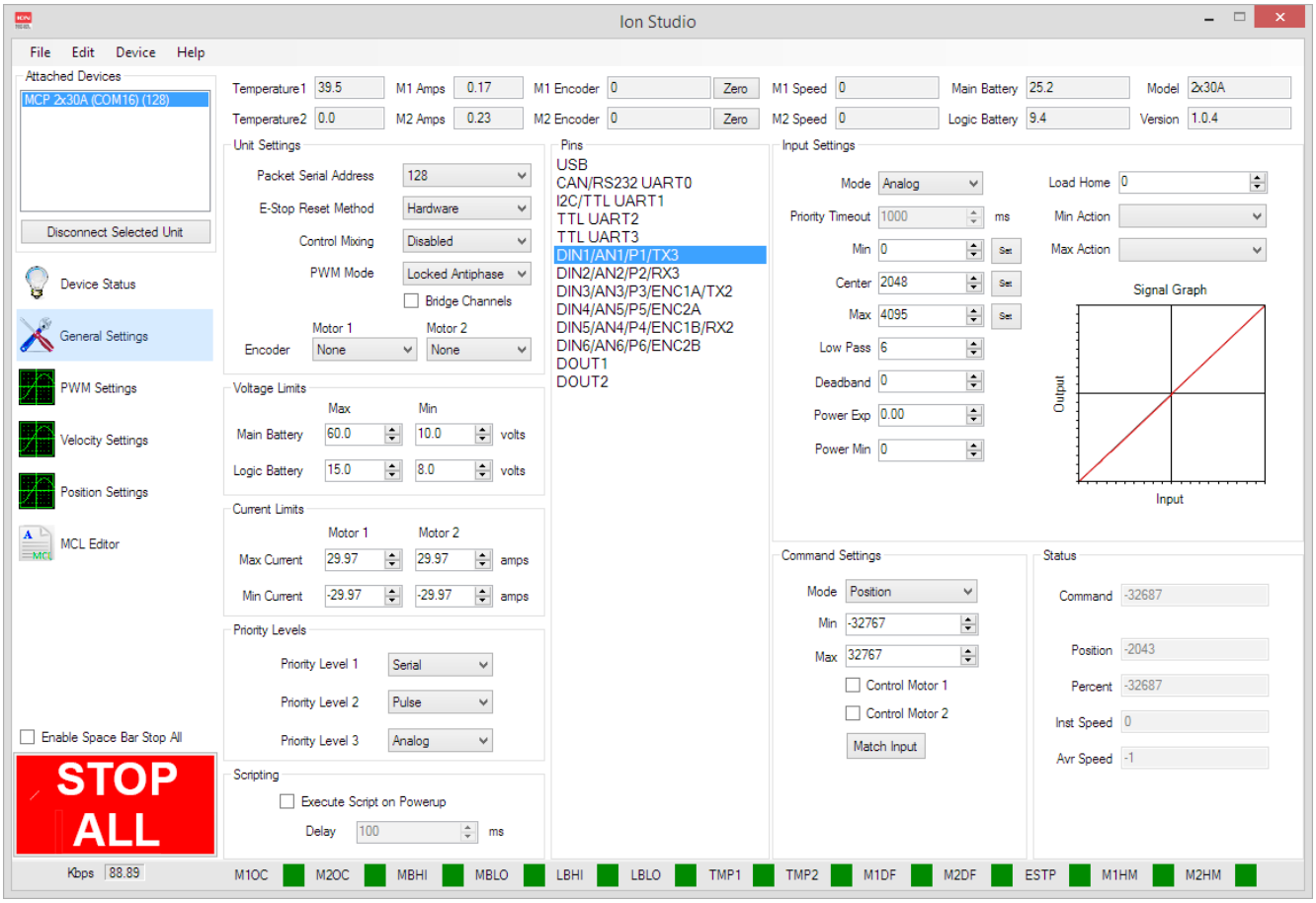
Control Motor 1 and 2

Enables the input singal to control the specified motor(s).

Match Input
The Match Input option calculates the input range with center being 0 and sets the Min and Max value range to match it. An example is Min was 1000, Center is 2000 and Max is 3000 it would set the command range Min / max to -1000 and +1000

1.9.12 Status

The status displays the current input signal parameters. The status display is updated continiously and can be used to verify the input signal.



Command
Displays the scaled command value based on the Command Min,Max range and Mode

Position
Displays the input signals value after ranging, filtering, deadband, Power Exp and Power Min offset have been calciulated.

Percent
Displays the command percent scaled to +/- 32,767

Instantaneous Speed
Displays the most recent speed reading (updated 625 times a second) of the input

Average Speed
Displays the 1 second running average speed of the input signal.

2.0 Communications

2.0.1 Communication and Input Types

The MCP offers several methods of communications. User generated MCL programs can access each type. During initial development USB is best suited due to its simplicity and speed. USB is not immune to electrically noisy environments and RS-232 serial or CANopen are recommended for the final application.

- **USB Packet Serial** - A USB connection can be used to control the MCP controller an interface to an MCL program. Ion Studio uses packet serial communications over USB to configure the controller.

- **RS-232 Packet Serial** - (+/- 12v) For high noise environments RS-232 communications may be used. RS-232 has a robust signaling voltage which prevents almost all forms of electrical noise from interfering with communications.

- **TTL Packet Serial** - (0v to 5v) Same as RS-232 without the level translator. This allows easy integration with devices such as the Arduino or Raspberry-Pi microcontrollers.

- **CANopen** - Open industry standard that allows multiple types of controls to be synchronously controlled from a single master controller.

- **Digital Signals** - Used for inputting limit and/or home switches as well as custom uses in user generated MCL programs.

- **Analog Signals** - Used for inputting analog voltages from many types of sensors for use as position encoders or in user generated MCL programs.

- **PWM Signals** - Inputs for different sensor types including magnetic linear and rotatory. Sensor can be used for position and velocity control or in user generated MCL programs.

- **RC Pulse Signals** - Control inputs from a R/C Radio receiver. Can be used for motor control or in user generated MCL programs.

- **Quadrature Pulse Signals** - Used for velocity and position control as well as for jog control or user generated MCL programs.

2.1 USB Communications

2.1.1 USB Connection

A USB connection can be used to control the MCP. The USB connection use packet serial commands. The USB connection can also interface to a running MCL script. The MCP can communicate over USB using packet serial commands at any time.

2.1.2 USB Power

The MCP USB port is self powered. This means it receives no power from the USB cable. The controller must be externally powered to function.

2.1.3 USB Comport and Baud rate

The MCP will be detected as a CDC Virtual Comport. When connected to a Windows PC, a driver must be installed. The driver is available for download from our website. On Linux or OSX the MCP will be automatically detected as a virtual comport and an appropriate driver will be automatically loaded.

Unlike a physical serial port the USB CDC Virtual Comport does not need a baud rate to be set correctly. It will always communicate at the fastest speed the master and slave device can reach. This will typically be around 1mb/s.

2.2 Packet Serial Mode

2.2.1 Packet Serial Communications

Packet serial is a buffered bidirectional serial mode. Sophisticated instructions can be sent to the controller. The basic command structures consist of an address byte, command byte, data bytes and a CRC16 16bit checksum. The amount of data each command will send or receive can vary.

2.2.2 Address

Packet serial requires a unique address. With up to 8 addresses available you can control up to 8 motor controllers on the same TTL serial port when properly wired. Addresses range form 0x80 to 0x87

2.2.3 Packet Serial Baud Rate

When in serial mode or packet serial mode the baud rate can be changed to any baud rate between 300 and 1000000bps

2.2.4 Packet Timeout

When sending a packet to the MCP, if there is a delay longer than 10ms between bytes being received in a packet, the MCP will discard the entire packet. This will allow the packet buffer to be cleared by simply adding a minimum 10ms delay before sending a new packet command in the case of a communications error. This can usually be accommodated by having a 10ms timeout when waiting for a reply from the MCP. If the reply times out the packet buffer will have been cleared automatically.

2.2.5 Packet Acknowledgement

MCP will send an acknowledgment byte on write only packet commands that are valid. The value sent back is 0xFF. If the packet was not valid for any reason no acknowledgement will be sent back.

2.2.6 CRC16 Checksum Calculation

MCP uses a CRC(Cyclic Redundancy Check) to validate each packet it receives. This is more complex than a simple checksum but prevents errors that could otherwise cause unexpected actions to execute on the controller.

The CRC can be calculated using the following code(example in C):

```
//Calculates CRC16 of nBytes of data in byte array message
unsigned int crc16(unsigned char *packet, int nBytes) {
    for (int byte = 0; byte < nBytes; byte++) {
        crc = crc ^ ((unsigned int)packet[byte] << 8);
        for (unsigned char bit = 0; bit < 8; bit++) {
            if (crc & 0x8000) {
                crc = (crc << 1) ^ 0x1021;
            } else {
                crc = crc << 1;
            }
        }
    }
    return crc;
}
```

2.2.7 CRC16 Checksum Calculation for Received data

The CRC16 calculation can also be used to validate data sent from the MCP. The CRC16 value should be calculated using the sent Address and Command byte as well as all the data received back from the Roboclaw except the two CRC16 bytes. The value calculated will match the CRC16 sent by the Roboclaw if there are no errors in the data sent or received.

2.2.8 Easy to use Libraries

Source code and Libraries are available on the Ion Motion Control website that already handle the complexities of using packet serial with the MCP. Libraries are available for Arduino(C++), C# on Windows(.NET) or Linux(Mono) and Python(Raspberry Pi, Linux, OSX, etc) as well as a LabView Instrument Driver.

2.2.9 Handling values larger than a byte

Many Packet Serial commands require values larger than a byte can hold. In order to send or receive those values they need to be broken up into 2 or more bytes. There are two ways this can be done, high byte first or low byte first. Roboclaw expects the high byte first. All command arguments and values are either single bytes, words (2 bytes) or longs (4 bytes). All arguments and values are integers (signed or unsigned). No floating point values (numbers with decimal places) are used in Packet Serial commands.

To convert a 32bit value into 4 bytes you just need to shift the bits around:

```
unsigned char byte3 = MyLongValue>>24; //High byte
unsigned char byte2 = MyLongValue>>16;
unsigned char byte1 = MyLongValue>>8;
unsigned char byte0 = MyLongValue;            //Low byte
```

The same applies to 16bit values:

```
unsigned char byte1 = MyWordValue>>8;  //High byte
unsigned char byte0 = MyWordValue;            //Low byte
```

The oposite can also be done. Convert several bytes into a 16bit or 32bit value:

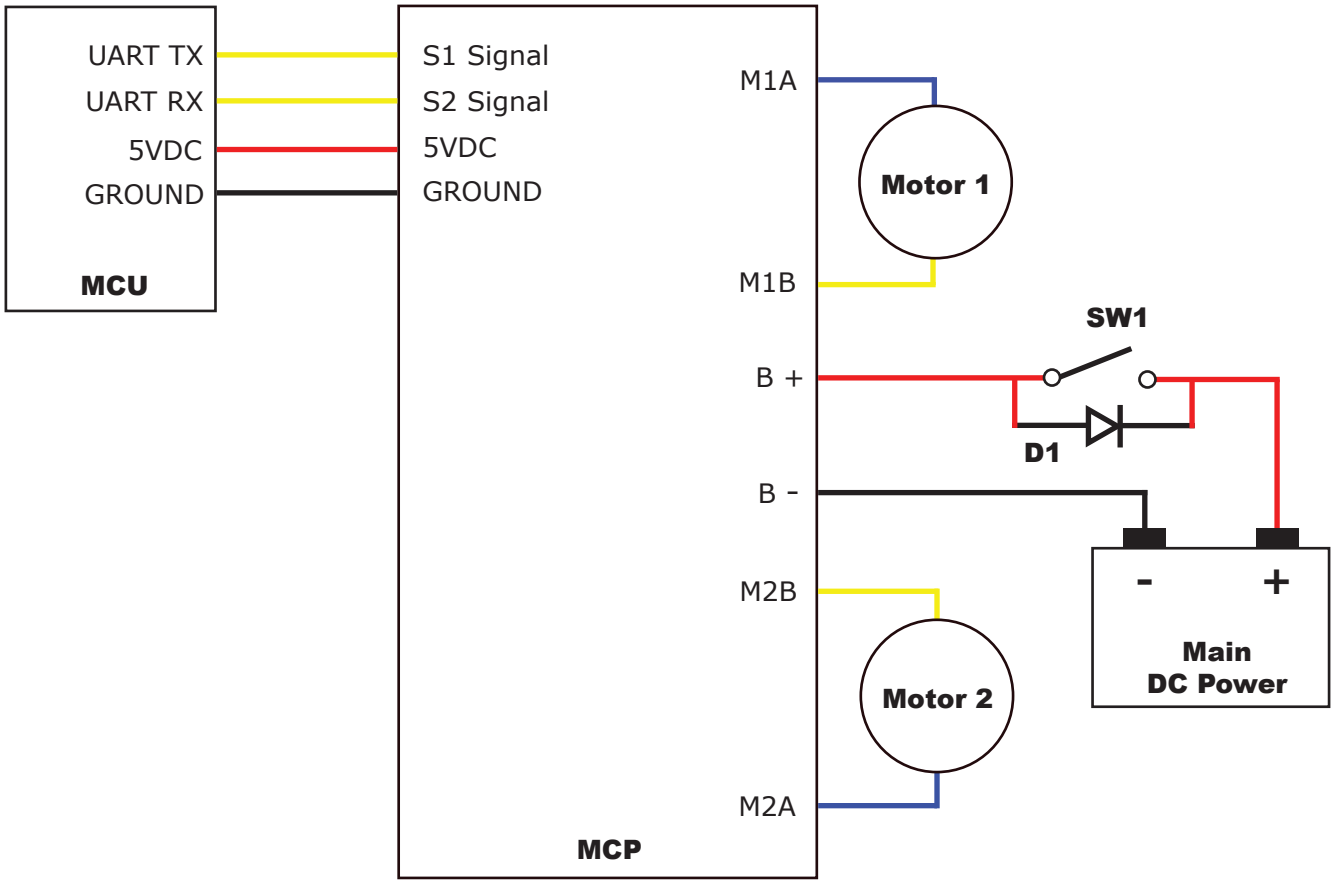
```
unsigned long MyLongValue = byte3<<24 | byte2<<16 | byte1<<8 | byte0;

unsigned int MyWordValue = byte1<<8 | byte0;
```

Packet Serial commands, when a value must be broken into multiple bytes or combined from multiple bytes it will be indicated either by (2 bytes) or (4 bytes).

2.2.10 Packet Serial Wiring

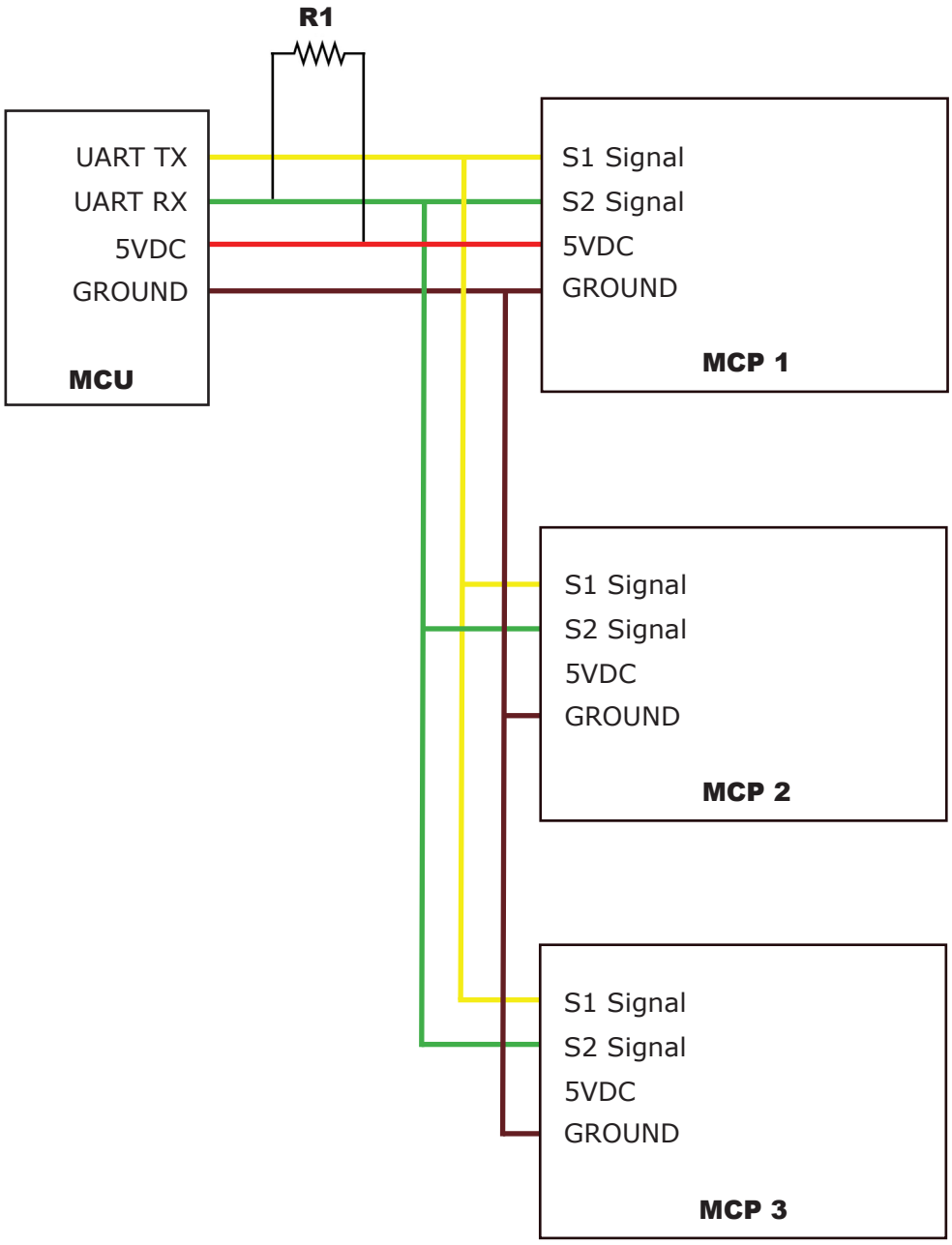
In packet serial mode the MCP can transmit and receive serial data. A microcontroller with a UART is recommended. The UART will buffer the data received from controller. When a request for data is made to the MCP the return data will have at least a 1ms delay after the command is received if the baudrate is set at or below 38400. This will allow slower processors and processors without UARTs to communicate with the controller.

The diagram below shows the main battery as the only power source. Make sure the LB jumper is set correctly. The 5VDC shown connected is only required if your MCU needs a power source. This is the BEC feature of the MCP. If the MCU has its own power source do not connect the 5VDC.



2.2.11 Multi-Unit Packet Serial Wiring

In packet serial mode up to eight MCP units can be controlled from a single serial port. The wiring diaghram below illustrates how this is done. Each controller must have multi-unit mode enabled and have a unique packet serial address set (see General Settings in IonMotion). Wire the S1 and S2 pins directly to the MCU TX and RX pins. Install a pullup resistor (R1) on the MCU RX pin. A 1K to 4.7K resistor value is recommended.



2.2.12 Commands 0 - 7 Compatibility Commands

The following commands are used in packet serial mode. The command syntax is the same for commands 0 thru 7:

```
Send: Address, Command, ByteValue, CRC16
Receive: [0xFF]
```

Command	Description
0	Drive Forward Motor 1
1	Drive Backwards Motor 1
2	Set Main Voltage Minimum
3	Set Main Voltage Maximum
4	Drive Forward Motor 2
5	Drive Backwards Motor 2
6	Drive Motor 1 (7 Bit)
7	Drive Motor 2 (7 Bit)
8	Drive Forward Mixed Mode
9	Drive Backwards Mixed Mode
10	Turn Right Mixed Mode
11	Turn Left Mixed Mode
12	Drive Forward or Backward (7 bit)
13	Turn Left or Right (7 Bit)

0 - Drive Forward M1

Drive motor 1 forward. Valid data range is 0 - 127. A value of 127 = full speed forward, 64 = about half speed forward and 0 = full stop.

```
Send: [Address, 0, Value, CRC(2 bytes)]
Receive: [0xFF]
```

1 - Drive Backwards M1

Drive motor 1 backwards. Valid data range is 0 - 127. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

```
Send: [Address, 1, Value, CRC(2 bytes)]
Receive: [0xFF]
```

2 - Set Minimum Main Voltage (Command 57 Preferred)

Sets main battery (B- / B+) minimum voltage level. If the battery voltages drops below the set voltage level RoboClaw will stop driving the motors. The voltage is set in .2 volt increments. A value of 0 sets the minimum value allowed which is 6V. The valid data range is 0 - 140 (6V - 34V). The formula for calculating the voltage is: (Desired Volts - 6) x 5 = Value. Examples of valid values are 6V = 0, 8V = 10 and 11V = 25.

```
Send: [Address, 2, Value, CRC(2 bytes)]
Receive: [0xFF]
```

3 - Set Maximum Main Voltage (Command 57 Preferred)
Sets main battery (B- / B+) maximum voltage level. The valid data range is 30 - 175 (6V - 34V). During regenerative breaking a back voltage is applied to charge the battery. When using a power supply, by setting the maximum voltage level, RoboClaw will, before exceeding it, go into hard braking mode until the voltage drops below the maximum value set. This will prevent overvoltage conditions when using power supplies. The formula for calculating the voltage is: Desired Volts x 5.12 = Value. Examples of valid values are 12V = 62, 16V = 82 and 24V = 123.

```
Send: [Address, 3, Value, CRC(2 bytes)]
Receive: [0xFF]
```

4 - Drive Forward M2
Drive motor 2 forward. Valid data range is 0 - 127. A value of 127 full speed forward, 64 = about half speed forward and 0 = full stop.

```
Send: [Address, 4, Value, CRC(2 bytes)]
Receive: [0xFF]
```

5 - Drive Backwards M2
Drive motor 2 backwards. Valid data range is 0 - 127. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

```
Send: [Address, 5, Value, CRC(2 bytes)]
Receive: [0xFF]
```

6 - Drive M1 (7 Bit)
Drive motor 1 forward or reverse. Valid data range is 0 - 127. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

```
Send: [Address, 6, Value, CRC(2 bytes)]
Receive: [0xFF]
```

7 - Drive M2 (7 Bit)
Drive motor 2 forward or reverse. Valid data range is 0 - 127. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

```
Send: [Address, 7, Value, CRC(2 bytes)]
Receive: [0xFF]
```

2.2.13 Commands 8 - 13 Mixed Mode Compatibility Commands

The following commands are mix mode compatibility commands used to control speed and turn using differential steering. Before a command is executed valid drive and turn data is required. You only need to send both data packets once. After receiving both valid drive and turn data RoboClaw will begin to operate the motors. At this point you only need to update turn or drive data as needed.

8 - Drive Forward

Drive forward in mix mode. Valid data range is 0 - 127. A value of 0 = full stop and 127 = full forward.

```
Send: [Address, 8, Value, CRC(2 bytes)]
Receive: [0xFF]
```

9 - Drive Backwards

Drive backwards in mix mode. Valid data range is 0 - 127. A value of 0 = full stop and 127 = full reverse.

```
Send: [Address, 9, Value, CRC(2 bytes)]
Receive: [0xFF]
```

10 - Turn right

Turn right in mix mode. Valid data range is 0 - 127. A value of 0 = stop turn and 127 = full speed turn.

```
Send: [Address, 10, Value, CRC(2 bytes)]
Receive: [0xFF]
```

11 - Turn left

Turn left in mix mode. Valid data range is 0 - 127. A value of 0 = stop turn and 127 = full speed turn.

```
Send: [Address, 11, Value, CRC(2 bytes)]
Receive: [0xFF]
```

12 - Drive Forward or Backward (7 Bit)

Drive forward or backwards. Valid data range is 0 - 127. A value of 0 = full backward, 64 = stop and 127 = full forward.

```
Send: [Address, 12, Value, CRC(2 bytes)]
Receive: [0xFF]
```

13 - Turn Left or Right (7 Bit)

Turn left or right. Valid data range is 0 - 127. A value of 0 = full left, 0 = stop turn and 127 = full right.

```
Send: [Address, 13, Value, CRC(2 bytes)]
Receive: [0xFF]
```

2.3 Advance Packet Serial Mode

2.3.1 Command List

The following commands are used to read board status, version information and set/read configuration values.

Command	Description
21	Read Firmware Version
24	Read Main Battery Voltage
25	Read Logic Battery Voltage
26	Set Minimum Logic Voltage Level
27	Set Maximum Logic Voltage Level
48	Read Motor PWMs
49	Read Motor Currents
57	Set Main Battery Voltages
58	Set Logic Battery Voltages
59	Read Main Battery Voltage Settings
60	Read Logic Battery Voltage Settings
80	Restore Defaults
82	Read Temperature
83	Read Temperature 2
90	Read Status
91	Read Encoder Modes
92	Set Motor 1 Encoder Mode
93	Set Motor 2 Encoder Mode
94	Write Settings to EEPROM
95	Read Settings from EEPROM
128	Set M1 Inductance and Resistance
129	Set M2 Inductance and Resistance
130	Get M1 Inductance and Resistance
131	Get M2 Inductance and Resistance
133	Set M1 Maximum Current
134	Set M2 Maximum Current
135	Read M1 Maximum Current
136	Read M2 Maximum Current
137	Set DOUT Action
138	Get DOUT Actions
139	Set Priority Levels
140	Get Priority Levels
141	Set Address and Mixed Flag
142	Get Mixed Flag
143	Set Signal Properties
144	Get All Signal Properies
145	Set Stream Properties

Command	Description
146	Get All Stream Properties
147	Get Signal Values
148	Set PWM Mode
149	Read PWM Mode
200	E-Stop Reset
201	Lock/Unlock E-Stop Reset
202	Get E-Stop Lock
246	Set Script Autorun Delay
247	Get Script Autorun Delay
248	Start Script
249	Stop Script
252	Read User NVM Word
253	Write User NVM Word

21 - Read Firmware Version
Read RoboClaw firmware version. Returns up to 48 bytes(depending on the Roboclaw model) and is terminated by a line feed character and a null character.

```
Send: [Address, 21]
Receive: ["MCP266 2x60A v1.0.0",10,0, CRC(2 bytes)]
```

The command will return up to 48 bytes. The return string includes the product name and firmware version. The return string is terminated with a line feed (10) and null (0) character.

24 - Read Main Battery Voltage Level
Read the main battery voltage level connected to B+ and B- terminals. The voltage is returned in 10ths of a volt(eg 300 = 30v).

```
Send: [Address, 24]
Receive: [Value(2 bytes), CRC(2 bytes)]
```

25 - Read Logic Battery Voltage Level
Read a logic battery voltage level connected to LB+ and LB- terminals. The voltage is returned in 10ths of a volt(eg 50 = 5v).

```
Send: [Address, 25]
Receive: [Value.Byte1, Value.Byte0, CRC(2 bytes)]
```

26 - Set Minimum Logic Voltage Level
Note: This command is included for backwards compatibility. We recommend you use command 58 instead.

Sets logic input (LB- / LB+) minimum voltage level. RoboClaw will shut down with an error if the voltage is below this level. The voltage is set in .2 volt increments. A value of 0 sets the minimum value allowed which is 6V. The valid data range is 0 - 140 (6V - 34V). The formula for calculating the voltage is: (Desired Volts - 6) x 5 = Value. Examples of valid values are 6V = 0, 8V = 10 and 11V = 25.

```
Send: [Address, 26, Value, CRC(2 bytes)]
Receive: [0xFF]
```

27 - Set Maximum Logic Voltage Level
Note: This command is included for backwards compatibility. We recommend you use command 58 instead.

Sets logic input (LB- / LB+) maximum voltage level. The valid data range is 30 - 175 (6V - 34V). RoboClaw will shutdown with an error if the voltage is above this level. The formula for calculating the voltage is: Desired Volts x 5.12 = Value. Examples of valid values are 12V = 62, 16V = 82 and 24V = 123.

```
Send: [Address, 27, Value, CRC(2 bytes)]
Receive: [0xFF]
```

48 - Read Motor PWM values
Read the current PWM output values for the motor channels. The values returned are +/-32767. The duty cycle percent is calculated by dividing the Value by 327.67.

```
Send: [Address, 48]
Receive: [M1 PWM(2 bytes), M2 PWM(2 bytes), CRC(2 bytes)]
```

49 - Read Motor Currents
Read the current draw from each motor in 10ma increments. The amps value is calculated by dividing the value by 100.

```
Send: [Address, 49]
Receive: [M1 Current(2 bytes), M2 Currrent(2 bytes), CRC(2 bytes)]
```

57 - Set Main Battery Voltages
Set the Main Battery Voltage cutoffs, Min and Max. Min and Max voltages are in 10th of a volt increments. Multiply the voltage to set by 10.

```
Send: [Address, 57, Min(2 bytes), Max(2bytes, CRC(2 bytes)]
Receive: [0xFF]
```

58 - Set Logic Battery Voltages
Set the Logic Battery Voltages cutoffs, Min and Max. Min and Max voltages are in 10th of a volt increments. Multiply the voltage to set by 10.

```
Send: [Address, 58, Min(2 bytes), Max(2bytes, CRC(2 bytes)]
Receive: [0xFF]
```

59 - Read Main Battery Voltage Settings
Read the Main Battery Voltage Settings. The voltage is calculated by dividing the value by 10

```
Send: [Address, 59]
Receive: [Min(2 bytes), Max(2 bytes), CRC(2 bytes)]
```

60 - Read Logic Battery Voltage Settings
Read the Logic Battery Voltage Settings. The voltage is calculated by dividing the value by 10

```
Send: [Address, 60]
Receive: [Min(2 bytes), Max(2 bytes), CRC(2 bytes)]
```

80 - Restore Defaults
Reset Settings to factory defaults.

```
Send: [Address, 80, CRC(2 bytes)]
Receive: [0xFF]
```

82 - Read Temperature
Read the board temperature. Value returned is in 10ths of degrees.

```
Send: [Address, 82]
Receive: [Temperature(2 bytes), CRC(2 bytes)]
```

83 - Read Temperature 2
Read the second board temperature(only on supported units). Value returned is in 10ths of degrees.

```
Send: [Address, 83]
Receive: [Temperature(2 bytes), CRC(2 bytes)]
```

90 - Read Status
Read the current unit status.

```
Send: [Address, 90]
Receive: [Status, CRC(2 bytes)]
```

Function	Status Bit Mask
Normal	0x0000
M1 OverCurrent Warning	0x0001
M2 OverCurrent Warning	0x0002
E-Stop	0x0004
Temperature Error	0x0008
Temperature2 Error	0x0010
Main Battery High Error	0x0020
Logic Battery High Error	0x0040
Logic Battery Low Error	0x0080
Main Battery High Warning	0x0400
Main Battery Low Warning	0x0800
Termperature Warning	0x1000
Temperature2 Warning	0x2000

91 - Read Encoder Mode
Read the encoder pins assigned for both motors.

```
Send: [Address, 91]
Receive: [Enc1Mode, Enc2Mode, CRC(2 bytes)]
```

92 - Set Motor 1 Encoder Mode
Set the Encoder Pin for motor 1. See command 91.

```
Send: [Address, 92, Pin, CRC(2 bytes)]
Receive: [0xFF]
```

93 - Set Motor 2 Encoder Mode
Set the Encoder Pin for motor 2. See command 91.

```
Send: [Address, 93, Pin, CRC(2 bytes)]
Receive: [0xFF]
```

94 - Write Settings to EEPROM
Writes all settings to non-volatile memory. Values will be loaded after each power up.

```
Send: [Address, 94]
Receive: [0xFF]
```

95 - Read Settings from EEPROM
Read all settings from non-volatile memory.

```
Send: [Address, 95]
Receive: [Enc1Mode, Enc2Mode, CRC(2 bytes)]
```

128 - Set M1 Inductance and Resistance
Set Motor Inductance and Resistance values. Used for torque mode control. Inductance = L * 16777216. Resistance = R * 16777216. L is in henrys and R is in ohms. For example a motor with L of .6 millihenries = .0006henrys = .0006 * 16777216 = 10066. A motor with resistance of 6mohm = .006 ohm = .006 * 16777216 = 100663.

```
Send: [Address, 128, Inductance(4 bytes), Resistance (4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

129 - Set M2 Inductance and Resistance
Set Motor Inductance and Resistance values. Used for torque mode control. See Command 128.

```
Send: [Address, 129, Inductance(4 bytes), Resistance (4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

130 - Get M1 Inductance and Resistance
Get Motor Inductance and Resistance values. Used for torque mode control. Caluclate actual motor L and R by dividing Inductance and Resistance by 16777216. For example if Inductance is 201123, actual L in Henrys is 201123/16777216 = .0012 Henrys.

```
Send: [Address, 130]
Receive: [Inductance(4 bytes), Resistance(4 bytes), CRC(2 bytes)]
```

131 - Get M2 Inductance and Resistance
Get Motor Inductance and Resistance values. Used for torque mode control. See Cmd 130

```
Send: [Address, 131]
Receive: [Inductance(4 bytes), Resistance(4 bytes), CRC(2 bytes)]
```

133 - Set M1 Max Current Limit
Set Motor 1 Maximum Current Limit. Current value is in 10ma units. To calculate multiply current limit by 100.

```
Send: [Address, 134, MaxCurrent(4 bytes), 0, 0, 0, 0, CRC(2 bytes)]
Receive: [0xFF]
```

134 - Set M2 Max Current Limit
Set Motor 2 Maximum Current Limit. Current value is in 10ma units. To calculate multiply current limit by 100.

```
Send: [Address, 134, MaxCurrent(4 bytes), 0, 0, 0, 0, CRC(2 bytes)]
Receive: [0xFF]
```

135 - Read M1 Max Current Limit

Read Motor 1 Maximum Current Limit. Current value is in 10ma units. To calculate divide value by 100. MinCurrent is always 0.

```
Send: [Address, 135]
Receive: [MaxCurrent(4 bytes), MinCurrent(4 bytes), CRC(2 bytes)]
```

136 - Read M2 Max Current Limit

Read Motor 2 Maximum Current Limit. Current value is in 10ma units. To calculate divide value by 100. MinCurrent is always 0.

```
Send: [Address, 136]
Receive: [MaxCurrent(4 bytes), MinCurrent(4 bytes), CRC(2 bytes)]
```

137 - Set DOUT Action

Sets Action that triggers a DOUT pin. For example if an Overvoltage is triggered DOUT pin 1 can be triggered to enable an external voltage dump circuit.

Actions	
Motor1 is active	0x01
Motor2 is active	0x02
Either motor is active	0x03
Motor1 is reversed	0x04
Motor2 is reversed	0x05
Either motor is reversed	0x06
Overvoltage	0x07
Overtemperature	0x08
Stat1 LED	0x09
Stat2 LED	0x0A
Err LED	0x0B

```
Send: [Address, 137, Index(1 byte), Action(1 bytes), CRC(2 bytes)]
```

138 - Get DOUT Actions

Get all DOUT actions. This command sends the action settings for all DOUT pins. The number of pins depends on the model of the unit. The first byte sent back indicates the number of DOUT pins for the unit. See Actions table in command 137.

```
Send: [Address, 138]
Receive: [Count(1 bytes), Actions(count bytes), CRC(2 bytes)]
```

139 - Set Priority Levels

Set Priority Level signal types. Each of three Priority Levels can have a control signal type selected. The higher priority level is used first. If that input is inactive(determine by a timeout value) the Priority Level is changed to the next level down. if the particular Priority level is disabled the next lower level is moved to. if a higher level signal input becomes active the Priority level is changed back to the higher level.

Priority Level Signals

Disabled	0
Packet Serial	1
Pulse/Pwm	2
Analog	3

```
Send: [Address, 139, Level1, Level2, Level3, CRC(2 bytes)]
Receive: [0xFF]
```

140 - Get Priority Levels

Get Priority Level settings. See command 139.

```
Send: [Address, 140]
Receive: [Level1, Level2, Level3, CRC(2 bytes)]
```

141 - Set Address and Mixed Flag

Set packet Serial Address and Mixed Mode flag. Address can be set to 0x80 to 0x87. The Mixed flag setting is 0 for disabled and 1 for enabled.

```
Send: [Address, 141, Address, Mixed, CRC(2 bytes)]
Receive: [0xFF]
```

142 - Get Mixed Flag

Get Mixed flag setting. See command 141

```
Send: [Address, 142]
Receive: [Mixed, CRC(2 bytes)]
```

143 - Set Signal Properties

Set Signal Properties. This command will set all the properties for one channel(eg P0,P1, PID1A, PID1B ect).

Types: (Note: Not all types are available on all input pins)

NONE	0
ANALOG	1
PULSE	2
ENCODER	3
PSERIAL	4

Modes:

Position	0
Percent(0 to 65535)	1
Average Speed	2
Speed	3

Target:

None	0
Motor1	1
Motor2	2
Both Motors	3

MinAction...MaxAction:

Motor1 safestop	0x0001
Motor2 safestop	0x0100
Both Motors safestop	0x0101
Motor1 off	0x0002
Motor2 off	0x0200
Both Motors off	0x0202
Motor1 reverse	0x0004
Motor2 reverse	0x0400
Both Motors reverse	0x0404
Motor1 Forward Limit	0x0008
Motor2 Forward Limit	0x0800
Both Motors Forward Limit	0x0808
Motor1 Reverse Limit	0x0010
Motor2 Reverse Limit	0x1000
Both Motors Reverse Limit	0x1010
Motor1 load Loadhome value	0x0020
Motor2 load loadhome value	0x2000
Both Motors load loadhome	0x2020
Run Script	0x0040
Stop Script	0x4000
Reset Script	0x0080
Trigger ESTOP	0x8000

Timeout
The timeout in millisecond for the signal to be considered idle. This is used to determine when a lower priority level should be switched to.

LoadHome
value to set the position to on a LoadHome action. This is usually used to zero a quadrature encoder input.

Minimum...Maximum
The minimum and maximum position values. Any values higher or lower than these settings will trigger a MinAction or MaxAction if set.
Center: The middle point of the control input.

Deadband
For input signals that have slop near their center point Deadband can be set to zero this area. It can also be used to provide a large area where there is no effect on the position value, centered on the Center position.

PowerExp
The exponent setting. Values from -8*65536 to 8*65536 A value of 0 provides a linear control inpug. Values larger then 0 will provide more control resolution near the endpoints of a control input, while values less then 0 increase the control resolution near the control inputs center point.

PowerMin
Sets an absolute minimum signal value. For exampel if PowerMin is set to 100 then a any signal value >= 0 will be offset to 100. Any control input < 0 will be offset to -100.

MinOutput...MaxOutput
When the signal input is used to command a motor channel the signal can be scaled to any range of output. The default setting is for the full PWM range of a motor channel, -2068 to +2068. For example in some cases it may be prefered to limit the reverse power range on a motor(such as on a wheeled vehicle). Reducing the negative range will reduce the maximum power that can be applied in reverse.

```
Send: [Address,
       143,
       Pin(1 byte),
       Type(1 byte),
       Mode (1 byte),
       Target (1 byte),
       MinAction (1 byte),
       MaxAction (1 byte),
       Timeout(4 bytes),
       LoadHome(4 bytes),
       Minimum(4 bytes),
       Maximum(4 bytes),
       Center(4 bytes),
       Deadband(4 bytes),
       PowerExp(4 bytes),
       MinOutput(4 bytes),
       MaxOutput(4 bytes),
       PowerMin(4 bytes),
       CRC(2 bytes)]
Receive: [0xFF]
```

144 - Get Signal Properties

Read all Signal Properties. Command returns the pin count plus the properties for all the pins the controller supports. See command 143 for descriptions of the properties.

```
Send: [Address, 144]
Receive: [   Count,
             Type1(1 byte),
             Mode1 (1 byte),
             Target1 (1 byte),
             MinAction1 (1 byte),
             MaxAction1 (1 byte),
             Timeout1(4 bytes),
             LoadHome1(4 bytes),
             Minimum1(4 bytes),
             Maximum1(4 bytes),
             Center1(4 bytes),
             Deadband1(4 bytes),
             PowerExp1(4 bytes),
             MinOutput1(4 bytes),
             MaxOutput1(4 bytes),
             PowerMin1(4 bytes),
             ...,
             TypeN(1 byte),
             ModeN(1 byte),
             TargetN(1 byte),
             MinActionN(1 byte),
             MaxActionN(1 byte),
             TimeoutN(4 bytes),
             LoadHomeN(4 bytes),
             MinimumN(4 bytes),
             MaximumN(4 bytes),
             CenterN(4 bytes),
             DeadbandN(4 bytes),
             PowerExpN(4 bytes),
             MinOutputN(4 bytes),
             MaxOutputN(4 bytes),
             PowerMinN(4 bytes),
             CRC(2 bytes)]
Receive: [0xFF]
```

145 - Set Stream Properties
Set Stream Properties.

Index:

USB	0
RS232 UART0	1
TTL UART1	2
TTL UART2	3
TTL UART3	4

Type:

MCL	0
PacketSerial	1
Disabled	2

(Note: CAN will be active if UART0 is disabled, I2C will be active if UART1 is disabled)

Rate: The bps setting for the stream

Timeout: the timeout in milliseconds before the stream is considered inactive.

```
Send: [Address, 145, Index(byte), Type(byte), Rate(4 bytes), timeout(4 bytes), CRC(2
bytes)]
Receive: [0xFF]
```

146 - Get Stream Properties
Read all stream properties. Returns the stream CNT followed by all stream properties.

```
Send: [Address, 146]
Receive: [streamcnt, type1(byte), rate1(4 bytes), timeout1(4 bytes),..., typeN, rateN,
timeoutN, CRC(2 bytes)]
```

147 - Get Signal Values
Get all signal values for all inputs

Count: The intput pin count on the unit.
Command: The value of the currently selected mode(see command 143 mode).
Position: The current signal position.
Percent: The current signal percent.
Speed: The current instant speed.
SpeedS: The current average speed.

```
Send: [Address, 147]
Receive: [Count(byte), Command1(4 bytes), Position1(4 bytes), Percent1 (4 bytes), Speed1 (4
bytes), SpeedS1 (4 bytes),...,  CommandN, PositionN, PercentN, SpeedN, SpeedSN, CRC(2
bytes)]
```

148 - Set PWM Mode
Set PWM Drive mode. Locked Antiphase(0) or Sign Magnitude(1).

```
Send: [Address, 148, Mode, CRC(2 bytes)]
Receive: [0xFF]
```

149 - Read PWM Mode
Read PWM Drive mode. See Command 148.

```
Send: [Address, 149]
Receive: [PWMMode, CRC(2 bytes)]
```

200 - E-Stop Reset
Reset an E-Stop error. This command will do nothing if the E-Stop is not first unlocked.

```
Send: [Address, 200, CRC(2 bytes)]
Receive: [0xFF]
```

201 - Lock/Unlock E-Stop Reset
Lock/Unlock the E-Stop reset capaibility. By default teh E-Stop software reset is locked.

Lock:

Unlock	0xAA
Lock	Any other value

```
Send: [Address, 201]
Receive: [Lock, CRC(2 bytes)]
```

202 - Get E-Stop Lock
Get Current E-Stop Reset Lock setting. See command 201.

```
Send: [Address, 202]
Receive: [Lock, CRC(2 bytes)]
```

246 - Set Script Autorun Delay
Set the delay, in milliseconds, after a reset before the currently loaded script runs. If the value is less than 100ms the script will not run(autorun is disabled).

```
Send: [Address, 246, Delay(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

247 - Get Script Autorun Delay
Get Autorun Delay

```
Send: [Address, 247]
Receive: [Delay(4 bytes), CRC(2 bytes)]
```

248 - Start Script
Start the loaded script

```
Send: [Address, 248, CRC(2 bytes)]
Receive: [0xFF]
```

249 - Stop Script
Stop the loaded Script.

```
Send: [Address, 249, CRC(2 bytes)]
Receive: [0xFF]
```

252 - Read User EEProm Word
Read a value from the User EEProm memory(256 bytes).

```
Send: [Address, 252, EEProm Address(byte)]
Receive: [Value(2 bytes), CRC(2 bytes)]
```

253 - Write User EEProm Word
Get Priority Levels.

```
Send: [Address, 253, Address(byte),Value(2 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

2.4 Encoders

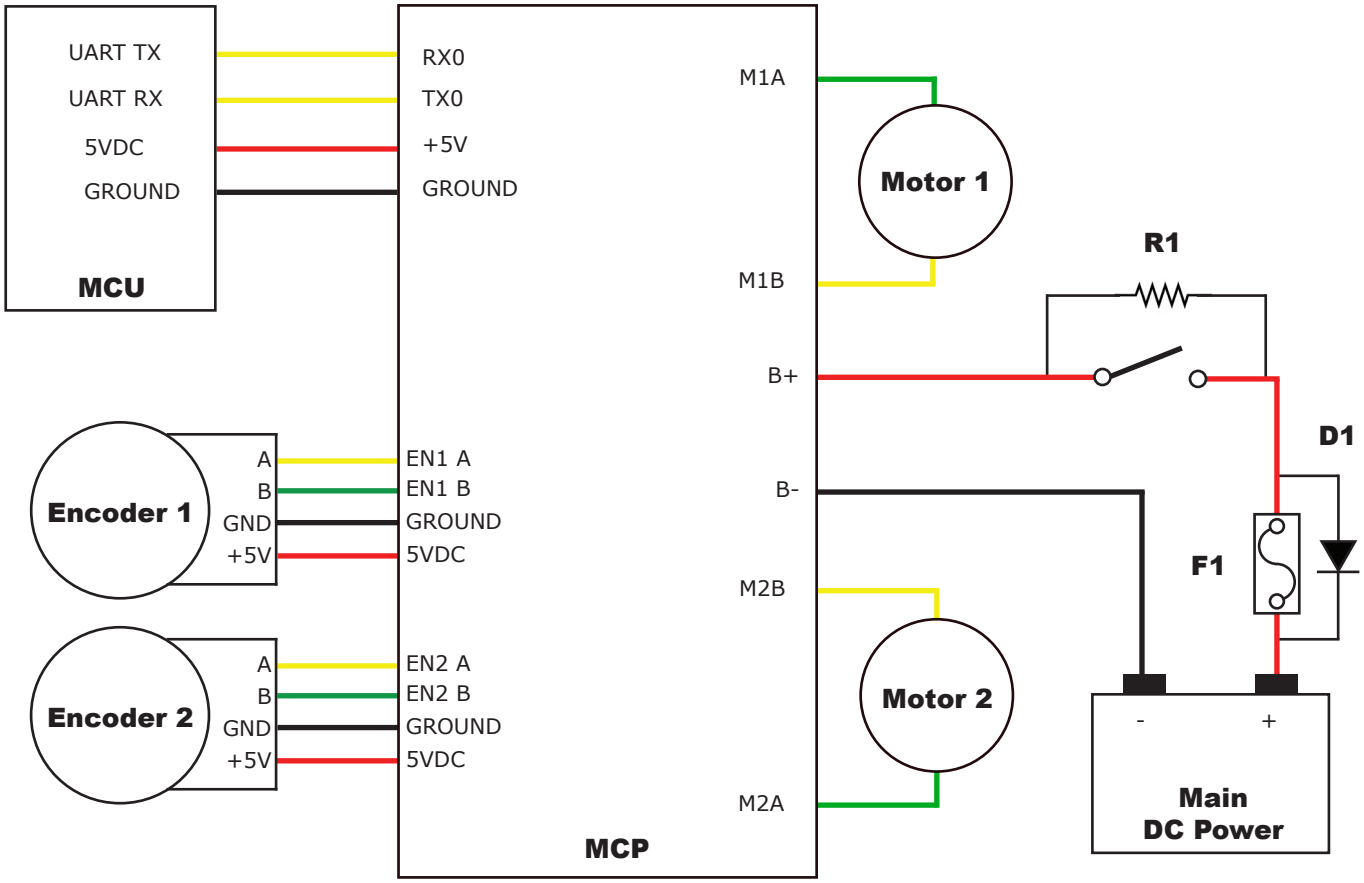
2.4.1 Cloosed Loop Modes

MCP supports a wide range of encoders in close loop mode. This section of the manual mainly deals with Quadrature and Absolute encoders. However additional types of encoders can be supported.

2.4.2 Quadrature Encoders

MCP is capable of reading two quadrature encoders, one for each motor channel. The main MCP interface provides a dual A and B input signals for each encoder as well as acces to regulated 5v.

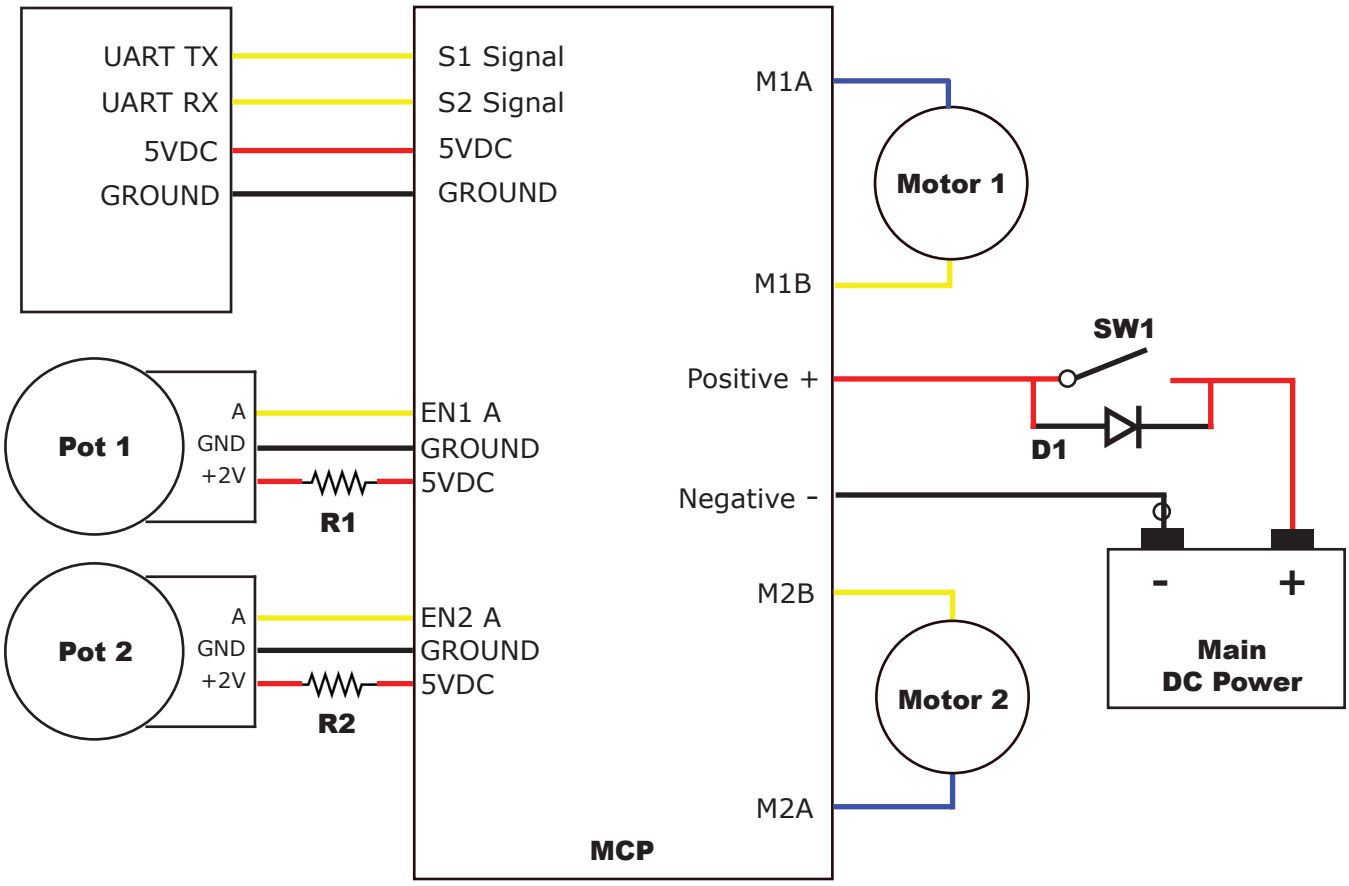
Quadrature encoders are directional. In a simple two motor robot, one motor will spin clock wise (CW) and the other motor will spin counter clock wise (CCW). The A and B inputs for one of the encoders must be reversed to allow both encoders to count up when the robot is moving forward. If both encoder are connected with leading edge pulse to channel A one will count up and the other down. This will cause commands like Mix Drive Forward to not work as expected. All motor and encoder combinations will need to be tuned.



2.4.3 Absolute Encoder

MCP is capable of reading absolute encoders that output an analog voltage. Like the Analog input modes for controlling the motors, the absolute encoder voltage must be between 0v and 5v.

The diagram below shows the main battery as the only power source. The 5VDC shown connected is only required if your MCU needs a power source. This is the BEC feature of the MCP. If the MCU has its own power source do not connect the 5VDC.



2.4.4 Encoder Tuning

To control motor speed and or position with an encoder the PID must be calibrated for the specific motor and encoder being used. Using Ion Studio the PID can be tuned manually or by the auto tune fucntion. Once the encoders are tuned the settings can be saved to the onboard eeprom and will be loaded each time the unit powers up.

The Ion Studio window for Velocity Settings will auto tune for velocity. The window for Position Settings can tune a simple PD position controller, a PID position controller or a cascaded Position with Velocity controller(PIV). The cascaded tune will determine both the velocity and position values for the motor but still requires the QPPS be manually set for the motor before starting. Auto tune functions usually return reasonable values but manually adjustments may be required for optimum performance.

2.4.5 Auto Tuning

Ion Studio provides the option to auto tune velocity and position control. To use the auto tune option make sure the encoder and motor are running in the desired direction and the basic PWM control of the motor works as expected. It is recommend to ensure the motor and encoder combination are functioning properly before using the auto tune feature.

1. Go to the PWM Settings screen in Ion Studio.

2. Slide the motor slider up to start moving the motor forward. Check the encoder is increasing in value. If it is not either reverse the motor wires or the encoder wires. The recheck.

Before using auto tune you must first set the motors and encoders maximum speed. For the purpose of auto tune the maximum quadrature pulse per second (QPPS) is the maximum speed the motor and encoder can acheive. When using an absolute encoder the QPPS will be the maximum rotational speed of the absolute encoder. Check the encoders data sheet to ensure the maximum rotational speed is not exceeded. Auto tune for position control can not automatically measure the maximum QPPS due to most position control systems having a limited range of movement.

3. To determine the maximum QPPS value, use the PWM settings screen to run the motor and encoder at 100% duty by moving the slider bar full up or down. Record the value from M1 Speed or M2 Speed fields at the top of the window. This is your maximum QPPS speed. If the motor can not be ran at full speed due to physical constraints, then an estimated maximum speed in encoder counts is required.

4. Enter the QPPS speed obtained from step 3 into the QPPS fields under settings. Ensure the correct QPPS is entered for the corresponding motor channel. Two identical motors and encoders may not function exactly the same so the maximum QPPS may vary.

5. To start auto tune click the auto tune button for the motor channel that is will be tuned first. The auto tune function will try to determine the best settings for that motor channel.

If the motor or encoder are wired incorrectly, the auto tune function can lock up and the motor controller will become unresponsive. Correct the wiring problem and reset the motor controller to continue.

2.4.6 Manual Velocity Calibration Procedure

1. Determine the quadrature pulses per second(QPPS) value for your motor. The simplest method to do this is to run the Motor at 100% duty using Ion Studio and read back the speed value from the encoder attached to the motor. If you are unable to run the motor like this due to physical constraints you will need to estimate the maximum speed in encoder counts the motor can produce.

2. Set the initial P,I and D values in the Velocity control window to 1,0 and 0. Try moving the motor using the slider controls in IonMotion. If the motor does not move it may not be wired correctly or the P value needs to be increased. If the motor immediately runs at max speed when you change the slider position you probably have the motor or encoder wires reversed. The motor is trying to go at the speed specified but the encoder reading is coming back in the opposite direction so the motor increases power until it eventually hits 100% power. Reverse the encoder or motor wires(not both) and test again.

3. Once the motor has some semblance of control you can set a moderate speed. Then start increasing the P value until the speed reading is near the set value. If the motor feels like it is vibrating at higher P values you should reduce the P value to about 2/3rds that value. Move on to the I setting.

4. Start increasing the I setting. You will usually want to increase this value by .1 increments. The I value helps the motor reach the exact speed specified. Too high an I value will also cause the motor to feel rough/vibrate. This is because the motor will over shoot the set speed and then the controller will reduce power to get the speed back down which will also under shoot and this will continue oscillating back and forth form too fast to too slow, causing a vibration in the motor.

5. Once P and I are set reasonably well usually you will leave D = 0. D is only required if you are unable to get reasonable speed control out of the motor using just P and I. D will help dampen P and I over shoot allowing higher P and I values, but D also increases noise in the calculation which can cause oscillations in the speed as well.

2.4.7 Manual Position Calibration Procedure

1. Position mode requires the Velocity mode QPPS value be set as described above. For simple Position control you can set Velocity P, I and D all to 0.

2. Set the Position I and D settings to 0. Set the P setting to 2000 as a reasonable starting point. To test the motor you must also set the Speed argument to some value. We recommend setting it to the same value as the QPPS setting(eg maximum motor speed). Set the minimum and maximum position values to safe numbers. If your motor has no dead stops this can be +-2 billion. If your motor has specific dead stops(like on a linear actuator) you will need to manually move the motor to its dead stops to determine these numbers. Leave some margin infront of each deadstop. Note that when using quadrature encoders you will need to home your motor on every power up since the quadrature readings are all relative to the starting position unless you set/reset the encoder values.

3. At this point the motor should move in the appropriate direction and stop, not necessarily close to the set position when you move the slider. Increase the P setting until the position is over shooting some each time you change the position slider. Now start increasing the D setting(leave I at 0). Increasing D will add dampening to the movement when getting close to the set position. This will help prevent the over shoot. D will usually be anywhere from 5 to 20 times larger than P but not always. Continue increasing P and D until the motor is working reasonably well. Once it is you have tuned a simple PD system.

4. Once your position control is acting relatively smoothly and coming close to the set position you can think about adjusting the I setting. Adding I will help reach the exact set point specified but in most motor systems there is enough slop in the gears that instead you will end up causing an oscillation around the specified position. This is called hunting. The I setting causes this when there is any slop in the motor/encoder/gear train. You can compensate some for this by adding deadzone. Deadzone is the area around the specified position the controller will consider to be equal to the position specified.

5. One more setting must be adjusted in order to use the I setting. The Imax value sets the maximum wind up allowed for the I setting calculation. Increasing Imax will allow I to affect a larger amount of the movement of the motor but will also allow the system to oscillate if used with a badly tuned I and/or set too high.

2.4.8 Encoder Commands

The following commands are used with the encoder(quadrature and absolute) hardware.

Command	Description
16	Read Encoder Count/Value for M1.
17	Read Encoder Count/Value for M2.
18	Read M1 Speed in Encoder Counts Per Second.
19	Read M2 Speed in Encoder Counts Per Second.
20	Resets Encoder Registers for M1 and M2(Quadrature only).
22	Set Encoder 1 Register(Quadrature only).
23	Set Encoder 2 Register(Quadrature only).
30	Read Current M1 Raw Speed
31	Read Current M2 Raw Speed
78	Read Encoders Counts
79	Read Motor Speeds

16 - Read Encoder Count/Value M1

Read M1 encoder count/position.

```
Send: [Address, 16]
Receive: [Enc1(4 bytes), Status, CRC(2 bytes)]
```

Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 4095 for the full 5.1v range.

The status byte tracks counter underflow, direction and overflow. The byte value represents:

Bit0 - Counter Underflow (1= Underflow Occurred, Clear After Reading)
Bit1 - Direction (0 = Forward, 1 = Backwards)
Bit2 - Counter Overflow (1= Underflow Occurred, Clear After Reading)
Bit3 - Reserved
Bit4 - Reserved
Bit5 - Reserved
Bit6 - Reserved
Bit7 - Reserved

17 - Read Quadrature Encoder Count/Value M2
Read M2 encoder count/position.

```
Send: [Address, 17]
Receive: [EncCnt(4 bytes), Status, CRC(2 bytes)]
```

Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 4095 for the full 5.1v range.

The Status byte tracks counter underflow, direction and overflow. The byte value represents:

Bit0 - Counter Underflow (1= Underflow Occurred, Cleared After Reading)
Bit1 - Direction (0 = Forward, 1 = Backwards)
Bit2 - Counter Overflow (1= Underflow Occurred, Cleared After Reading)
Bit3 - Reserved
Bit4 - Reserved
Bit5 - Reserved
Bit6 - Reserved
Bit7 - Reserved

18 - Read Encoder Speed M1
Read M1 counter speed. Returned value is in pulses per second. MCP keeps track of how many pulses received per second for both encoder channels.

```
Send: [Address, 18]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

Status indicates the direction (0 – forward, 1 - backward).

19 - Read Encoder Speed M2
Read M2 counter speed. Returned value is in pulses per second. MCP keeps track of how many pulses received per second for both encoder channels.

```
Send: [Address, 19]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

Status indicates the direction (0 – forward, 1 - backward).

20 - Reset Quadrature Encoder Counters
Will reset both quadrature decoder counters to zero. This command applies to quadrature encoders only.

```
Send: [Address, 20, CRC(2 bytes)]
Receive: [0xFF]
```

22 - Set Quadrature Encoder 1 Value
Set the value of the Encoder 1 register. Useful when homing motor 1. This command applies to quadrature encoders only.

```
Send: [Address, 22, Value(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

23 - Set Quadrature Encoder 2 Value
Set the value of the Encoder 2 register. Useful when homing motor 2. This command applies to quadrature encoders only.

```
Send: [Address, 23, Value(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

30 - Read Raw Speed M1
Read the pulses counted in that last 300th of a second. This is an unfiltered version of command 18. Command 30 can be used to make a independent PID routine. Value returned is in encoder counts per second.

```
Send: [Address, 30]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

The Status byte is direction (0 – forward, 1 - backward).

31 - Read Raw Speed M2
Read the pulses counted in that last 300th of a second. This is an unfiltered version of command 19. Command 31 can be used to make a independent PID routine. Value returned is in encoder counts per second.

```
Send: [Address, 31]
Receive: [Speed(4 bytes), Status, CRC(2 bytes)]
```

The Status byte is direction (0 – forward, 1 - backward).

78 - Read Encoder Counters
Read M1 and M2 encoder counters. Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 2047 for the full 2V analog range.

```
Send: [Address, 78]
Receive: [Enc1(4 bytes), Enc2(4 bytes), CRC(2 bytes)]
```

79 - Read ISpeeds Counters
Read M1 and M2 instantaneous speeds. Returns the speed in encoder counts per second for the last 300th of a second for both encoder channels.

```
Send: [Address, 79]
Receive: [ISpeed1(4 bytes), ISpeed2(4 bytes), CRC(2 bytes)]
```

2.4.9 Advanced Motor Control

The following commands are used to control motor speeds, acceleration distance and position using encoders.

Command	Description
28	Set Velocity PID Constants for M1.
29	Set Velocity PID Constants for M2.
32	Drive M1 With Signed Duty Cycle. (Encoders not required)
33	Drive M2 With Signed Duty Cycle. (Encoders not required)
34	Drive M1 / M2 With Signed Duty Cycle. (Encoders not required)
35	Drive M1 With Signed Speed.
36	Drive M2 With Signed Speed.
37	Drive M1 / M2 With Signed Speed.
38	Drive M1 With Signed Speed And Acceleration.
39	Drive M2 With Signed Speed And Acceleration.
40	Drive M1 / M2 With Signed Speed And Acceleration.
41	Drive M1 With Signed Speed And Distance. Buffered.
42	Drive M2 With Signed Speed And Distance. Buffered.
43	Drive M1 / M2 With Signed Speed And Distance. Buffered.
44	Drive M1 With Signed Speed, Acceleration and Distance. Buffered.
45	Drive M2 With Signed Speed, Acceleration and Distance. Buffered.
46	Drive M1 / M2 With Signed Speed, Acceleration And Distance. Buffered.
47	Read Buffer Length.
50	Drive M1 / M2 With Individual Signed Speed and Acceleration
51	Drive M1 / M2 With Individual Signed Speed, Accel and Distance
52	Drive M1 With Signed Duty and Accel. (Encoders not required)
53	Drive M2 With Signed Duty and Accel. (Encoders not required)
54	Drive M1 / M2 With Signed Duty and Accel. (Encoders not required)
55	Read Motor 1 Velocity PID Constants
56	Read Motor 2 Velocity PID Constants
61	Set Position PID Constants for M1.
62	Set Position PID Constants for M2
63	Read Motor 1 Position PID Constants
64	Read Motor 2 Position PID Constants
65	Drive M1 with Speed, Accel, Deccel and Position
66	Drive M2 with Speed, Accel, Deccel and Position
67	Drive M1 / M2 with Speed, Accel, Deccel and Position
68	Set Default Duty Acceleration for M1
69	Set Default Duty Acceleration for M2
81	Read Default Duty Acceleration Settings

28 - Set Velocity PID Constants M1
Several motor and quadrature combinations can be used with RoboClaw. In some cases the default PID values will need to be tuned for the systems being driven. This gives greater flexibility in what motor and encoder combinations can be used. The RoboClaw PID system consist of four constants starting with QPPS, P = Proportional, I= Integral and D= Derivative. The defaults values are:

```
QPPS = 44000
P = 0x00010000
I = 0x00008000
D = 0x00004000
```

QPPS is the speed of the encoder when the motor is at 100% power. P, I, D are the default values used after a reset. Command syntax:

```
Send: [Address, 28, D(4 bytes), P(4 bytes), I(4 bytes), QPPS(4 byte), CRC(2 bytes)]
Receive: [0xFF]
```

29 - Set Velocity PID Constants M2
Several motor and quadrature combinations can be used with RoboClaw. In some cases the default PID values will need to be tuned for the systems being driven. This gives greater flexibility in what motor and encoder combinations can be used. The RoboClaw PID system consist of four constants starting with QPPS, P = Proportional, I= Integral and D= Derivative. The defaults values are:

```
QPPS = 44000
P = 0x00010000
I = 0x00008000
D = 0x00004000
```

QPPS is the speed of the encoder when the motor is at 100% power. P, I, D are the default values used after a reset. Command syntax:

```
Send: [Address, 29, D(4 bytes), P(4 bytes), I(4 bytes), QPPS(4 byte), CRC(2 bytes)]Receive:
[0xFF]
```

32 - Drive M1 With Signed Duty Cycle
Drive M1 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder.

```
Send: [Address, 32, Duty(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32767 to +32767 (eg. +-100% duty).

33 - Drive M2 With Signed Duty Cycle
Drive M2 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder. The command syntax:

```
Send: [Address, 33, Duty(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty).

34 - Drive M1 / M2 With Signed Duty Cycle
Drive both M1 and M2 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder. The command syntax:

```
Send: [Address, 34, DutyM1(2 Bytes), DutyM2(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty).

35 - Drive M1 With Signed Speed
Drive M1 using a speed value. The sign indicates which direction the motor will turn. This command is used to drive the motor by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate as fast as possible until the defined rate is reached.

```
Send: [Address, 35, Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

36 - Drive M2 With Signed Speed
Drive M2 with a speed value. The sign indicates which direction the motor will turn. This command is used to drive the motor by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent, the motor will begin to accelerate as fast as possible until the rate defined is reached.

```
Send: [Address, 36, Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

37 - Drive M1 / M2 With Signed Speed
Drive M1 and M2 in the same command using a signed speed value. The sign indicates which direction the motor will turn. This command is used to drive both motors by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate as fast as possible until the rate defined is reached.

```
Send: [Address, 37, SpeedM1(4 Bytes), SpeedM2(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

38 - Drive M1 With Signed Speed And Acceleration
Drive M1 with a signed speed and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 38, Accel(4 Bytes), Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

39 - Drive M2 With Signed Speed And Acceleration
Drive M2 with a signed speed and acceleration value. The sign indicates which direction the motor will run. The acceleration value is not signed. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 39, Accel(4 Bytes), Speed(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

40 - Drive M1 / M2 With Signed Speed And Acceleration
Drive M1 and M2 in the same command using one value for acceleration and two signed speed values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. The motors are sync during acceleration. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 40, Accel(4 Bytes), SpeedM1(4 Bytes), SpeedM2(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

41 - Buffered M1 Drive With Signed Speed And Distance
Drive M1 with a signed speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. This command is used to control the top speed and total distance traveled by the motor. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 41, Speed(4 Bytes), Distance(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

42 - Buffered M2 Drive With Signed Speed And Distance
Drive M2 with a speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 42, Speed(4 Bytes), Distance(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

43 - Buffered Drive M1 / M2 With Signed Speed And Distance
Drive M1 and M2 with a speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 43, SpeedM1(4 Bytes), DistanceM1(4 Bytes),
             SpeedM2(4 Bytes), DistanceM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

44 - Buffered M1 Drive With Signed Speed, Accel And Distance
Drive M1 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control the motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 44, Accel(4 bytes), Speed(4 Bytes), Distance(4 Bytes),
            Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

45 - Buffered M2 Drive With Signed Speed, Accel And Distance
Drive M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control the motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 45, Accel(4 bytes), Speed(4 Bytes), Distance(4 Bytes),
            Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

46 - Buffered Drive M1 / M2 With Signed Speed, Accel And Distance
Drive M1 and M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control both motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 46, Accel(4 Bytes), SpeedM1(4 Bytes), DistanceM1(4 Bytes),
SpeedM2(4 bytes), DistanceM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

47 - Read Buffer Length
Read both motor M1 and M2 buffer lengths. This command can be used to determine how many commands are waiting to execute.

```
Send: [Address, 47]
Receive: [BufferM1, BufferM2, CRC(2 bytes)]
```

The return values represent how many commands per buffer are waiting to be executed. The maximum buffer size per motor is 64 commands(0x3F). A return value of 0x80(128) indicates the buffer is empty. A return value of 0 indiciates the last command sent is executing. A value of 0x80 indicates the last command buffered has finished.

50 - Drive M1 / M2 With Signed Speed And Individual Acceleration
Drive M1 and M2 in the same command using one value for acceleration and two signed speed values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. The motors are sync during acceleration. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

```
Send: [Address, 50, AccelM1(4 Bytes), SpeedM1(4 Bytes), AccelM2(4 Bytes),
      SpeedM2(4 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

51 - Buffered Drive M1 / M2 With Signed Speed, Individual Accel And Distance
Drive M1 and M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control both motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

```
Send: [Address, 51, AccelM1(4 Bytes), SpeedM1(4 Bytes), DistanceM1(4 Bytes),
AccelM2(4 Bytes), SpeedM2(4 bytes), DistanceM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

52 - Drive M1 With Signed Duty And Acceleration
Drive M1 with a signed duty and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by PWM and using an acceleration value for ramping. Accel is the rate per second at which the duty changes from the current duty to the specified duty.

```
Send: [Address, 52, Duty(2 bytes), Accel(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767(eg. +-100% duty). The accel value range is 0 to 655359(eg maximum acceleration rate is -100% to 100% in 100ms).

53 - Drive M2 With Signed Duty And Acceleration
Drive M1 with a signed duty and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by PWM and using an acceleration value for ramping. Accel is the rate at which the duty changes from the current duty to the specified dury.

```
Send: [Address, 53, Duty(2 bytes), Accel(2 Bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty). The accel value range is 0 to 655359 (eg maximum acceleration rate is -100% to 100% in 100ms).

54 - Drive M1 / M2 With Signed Duty And Acceleration
Drive M1 and M2 in the same command using acceleration and duty values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. This command is used to drive the motor by PWM using an acceleration value for ramping. The command syntax:

```
Send: [Address, CMD, DutyM1(2 bytes), AccelM1(4 Bytes), DutyM2(2 bytes),
      AccelM1(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty). The accel value range is 0 to 655359 (eg maximum acceleration rate is -100% to 100% in 100ms).

55 - Read Motor 1 Velocity PID and QPPS Settings
Read the PID and QPPS Settings.

```
Send: [Address, 55]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), QPPS(4 byte), CRC(2 bytes)]
```

56 - Read Motor 2 Velocity PID and QPPS Settings
Read the PID and QPPS Settings.

```
Send: [Address, 56]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), QPPS(4 byte), CRC(2 bytes)]
```

61 - Set Motor 1 Position PID Constants
The RoboClaw Position PID system consist of seven constants starting with P = Proportional, I= Integral and D= Derivative, MaxI = Maximum Integral windup, Deadzone in encoder counts, MinPos = Minimum Position and MaxPos = Maximum Position. The defaults values are all zero.

```
Send: [Address, 61, D(4 bytes), P(4 bytes), I(4 bytes), MaxI(4 bytes),
Deadzone(4 bytes), MinPos(4 bytes), MaxPos(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

Position constants are used only with the Position commands, 65,66 and 67 or when encoders are enabled in RC/Analog modes.

62 - Set Motor 2 Position PID Constants
The RoboClaw Position PID system consist of seven constants starting with P = Proportional, I= Integral and D= Derivative, MaxI = Maximum Integral windup, Deadzone in encoder counts, MinPos = Minimum Position and MaxPos = Maximum Position. The defaults values are all zero.

```
Send: [Address, 62, D(4 bytes), P(4 bytes), I(4 bytes), MaxI(4 bytes),
Deadzone(4 bytes), MinPos(4 bytes), MaxPos(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

Position constants are used only with the Position commands, 65,66 and 67 or when encoders are enabled in RC/Analog modes.

63 - Read Motor 1 Position PID Constants
Read the Position PID Settings.

```
Send: [Address, 63]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), MaxI(4 byte), Deadzone(4 byte),
             MinPos(4 byte), MaxPos(4 byte), CRC(2 bytes)]
```

64 - Read Motor 2 Position PID Constants
Read the Position PID Settings.

```
Send: [Address, 64]
Receive: [P(4 bytes), I(4 bytes), D(4 bytes), MaxI(4 byte), Deadzone(4 byte),
             MinPos(4 byte), MaxPos(4 byte), CRC(2 bytes)]
```

65 - Buffered Drive M1 with signed Speed, Accel, Deccel and Position
Move M1 position from the current position to the specified new position and hold the new position. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

```
Send: [Address, 65, Accel(4 bytes), Speed(4 Bytes), Deccel(4 bytes),
      Position(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

66 - Buffered Drive M2 with signed Speed, Accel, Deccel and Position
Move M2 position from the current position to the specified new position and hold the new position. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

```
Send: [Address, 66, Accel(4 bytes), Speed(4 Bytes), Deccel(4 bytes),
      Position(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

67 - Buffered Drive M1 & M2 with signed Speed, Accel, Deccel and Position
Move M1 & M2 positions from their current positions to the specified new positions and hold the new positions. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

```
Send: [Address, 67, AccelM1(4 bytes), SpeedM1(4 Bytes), DeccelM1(4 bytes),
             PositionM1(4 Bytes), AccelM2(4 bytes), SpeedM2(4 Bytes), DeccelM2(4 bytes),
                   PositionM2(4 Bytes), Buffer, CRC(2 bytes)]
Receive: [0xFF]
```

68 - Set M1 Default Duty Acceleration
Set the default acceleration for M1 when using duty cycle commands(Cmds 32,33 and 34) or when using Standard Serial, RC and Analog PWM modes.

```
Send: [Address, 68, Accel(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

69 - Set M2 Default Duty Acceleration

Set the default acceleration for M2 when using duty cycle commands(Cmds 32,33 and 34) or when using Standard Serial, RC and Analog PWM modes.

```
Send: [Address, 69, Accel(4 bytes), CRC(2 bytes)]
Receive: [0xFF]
```

81 - Read Default Duty Acceleration Settings

Read M1 and M2 Duty Cycle Acceleration Settings.

```
Send: [Address, 81]
Receive: [M1Accel(4 bytes), M2Accel(4 bytes), CRC(2 bytes)]
```

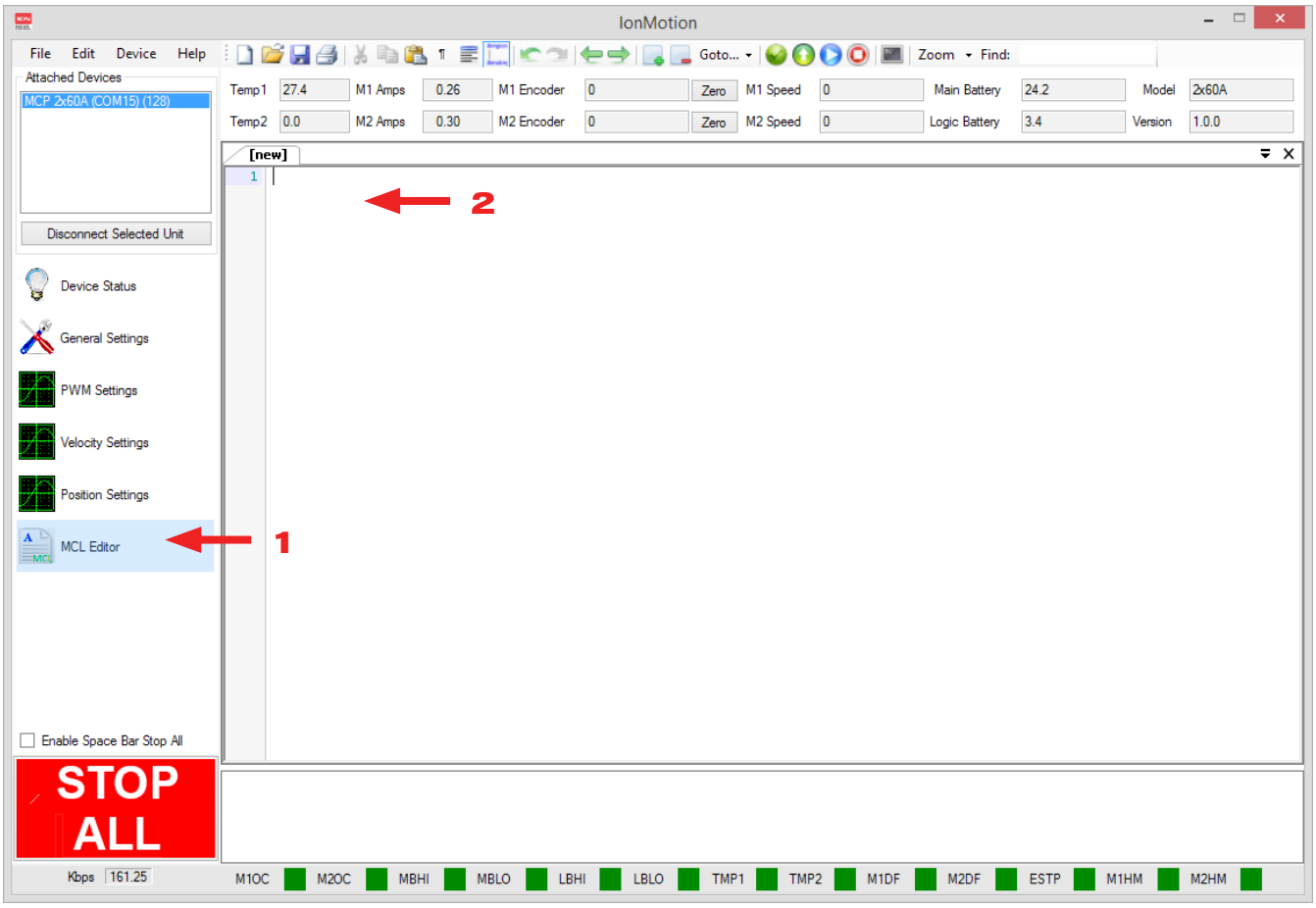
3.0 Programming MCP

3.0.1 MCL

The MCL programming language is based on a subset of BASIC. The MCL language provides full control over sensors and motor performance characteristics to create robust stand alone applications. MCL is capable of 32 bit integer and floating point math.

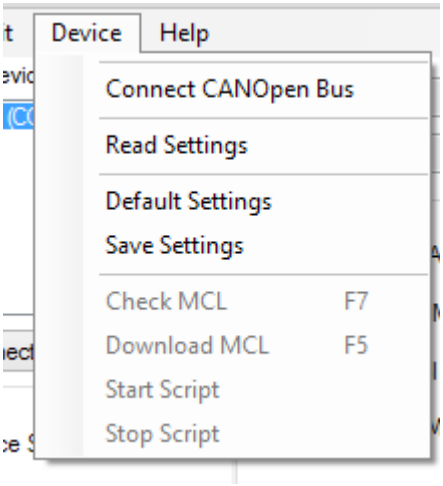
3.0.2 MCL Editor

Ion Studio incorporates an editor for creating and managing MCL programs. The editor is also used to comppile and downloaded user generated MCL programs to the MCP. To begin creating a new MCL program select the MCL Editor screen (1). Once the screen is open click in the editor window (2).



3.0.3 Device Menu

The Ion Studio **Device** menu is used to check and download a MCL program. After creating a MCL program use the **Check MCL** to compile and check the program for syntax errors. Once the program is checked, select **Download MCL** to send the program to the connect MCP.



3.0.4 Start and Stop Program Execution

Once a program is downloaded to the MCP and Ion Studio is connected use the **Start Script** to begin program execution. The **Stop Script** function can be used to halt program execution while Ion Studio is connected.

3.1 MCL Language

3.1.1 Variables

Variables are used to store values used in your program. The values stored will typically change during program run time. What was stored at the start of your program in a given variable can change by the time your program stops running.

To use a variable it must be defined in your program so the MCP knows to set aside the required amount of space in RAM. The MCP can handle up to 32 bit variables.

When creating a variable you specify what type of variable you will need in your program. You can define as many variables as you want. The only limiting factor is the amount of RAM available which varies for each MCP model.

3.1.2 Variable Types

Type	Bits	Value Range
Bit	1	1 or 0
Nib	4	0 to 15
Byte	8	0 to 255
SByte	8	-128 to +127
Word	16	0 to 65,535
SWord	32	-32,768 to +32,767
Long	32	0 to 4,294,967,295
SLong	32	-2,147,483,647 to +2,147,483,648
Float	32	± 2.0 EXP -126 to ± 2.0 EXP 127

3.1.3 Variable Locations

Where the variable is defined dictates where the variables can be used. Variables must be defined before they can be used. It is recommended to define your variables at the begining of your program.

3.1.4 Defining Variables

Variables are defined using the statement VAR. You can declare your new variable as any type found in the Variable Types table. To define a variable use the syntax shown below.

Syntax:

```
VariableName VAR Type
```

Examples:

```
Red VAR Byte
Tick VAR Nib
Switch var Bit
Totals var Sword
```

3.1.5 Variable Names

Variable names must start with a letter but can be any combination of letters and numbers including some symbols that are not in the reserved word list. You can name your variables almost anything. However, its a good idea to name them something useful that you will remember and understand for debugging your program. Variable names can be up to 1,024 characters long. The only names you can not use for your variables are reserved words. These are words used by Studio for commands or other syntax based names (see the manual appendix for a complete reserved word list).

Example:

```
MotorSensor VAR Word
```

3.1.6 Aliases

Variables can be aliased. Using an alias allows you to rename a variable in various places in your code. This helps your program be human readable without wasting additional memory space. Aliasing can also be used to access just a proportion of the variable such as the low or high byte for a word sized variable.

Example:

```
MotorDirection VAR Word
MaxCCSpeed VAR MotorDirection
MaxCCWSpeed VAR MotorDirection.byte0
```

3.1.7 Variable Modifiers

Variable modifiers can be used when only part of a variables value is needed. Most communication formats such as serial or I2C are byte driven. If you have a word sized variable and you are sending data to such a device, you will need to send one byte at a time of the word variable. The word value can be split using an alias with a modifier as shown below.

Example

```
MyData VAR Word
FirstByte VAR MyData.HighByte
SecondByte VAR MyData.LowByte
```

Variable modifiers can also be used inline with the variable in an expression.

Example

```
MyData VAR Word
MyByteSum var word

MyData = 0x1234
MyByteSum = MyData.byte1 + MyData.byte0 ;MyByteSum will equal 0x12 + 0x34
end
```

3.1.8 Variable Modifier Types

Modifier	Description
LowBit	Returns the low bit of a variable (least significant bit).
HighBit	Returns the high bit of a variable (most significant bit).
Bit*n*	Returns the *N*th bit of a variable. From 0 to 31 depending on variable size.
LowNib	Returns the low nibble (4 bits) of a variable (least significant nib).
HighNib	Returns the high nibble (4 bits) of a variable (most significant nib).
Nib*n*	Returns the *N*th nib (4 bits) of a variable. From 0 to 7 depending on variable size.
LowByte	Returns the low byte (8 bits) of a variable (least significant byte).
HighByte	Returns the high byte (8 bits) of a variable (most significant byte).
Byte*n*	Returns the *N*th byte of a variable. From 0 to 3 depending on variable size.
SByte*n*	Returns the *N*th signed byte of a variable. From 0 to 3 depending on variable size.
LowWord	Returns the low word (8 bits) of a variable (least significant word).
HighWord	Returns the high word (8 bits) of a variable (most significant word)
Word*n*	Returns the *N*th word of a variable. 0 to 1 depending on variable size.
SWord*n*	Returns the *N*th signed word of a variable. 0 to 1 depending on variable size.

3.1.9 Variable Arrays

Variable arrays can be used to store several values using one variable name. A good use of variable arrays would be taking multiple readings from a temperature sensor and storing the samples in a variable named Temperature. The reads can later be averaged for a more accurate temperature. MCL only supports one dimensional arrays.

Example:

```
Temperature VAR Byte(5)
```

The example above creates 5 byte sized RAM locations for the variable temperature. Each location is indexed and referenced by using a numerical value. The range is based on the array size that was defined. In this case the array size is 5. So an index value of 0 to 4 can be used.

Example:

```
Temperature(0) = 255 ;loads first position
Temperature(1) = 255 ;loads second position
Temperature(2) = 255 ;loads third position
Temperature(3) = 255 ;loads fourth position
Temperature(4) = 255 ;loads fifth position

Temp = Temperature(0) ;Loads value from first position in the array
```

A common use of variable arrays is to store strings of ASCII based characters. This can be used to send an entire sentence of text to a byte driven device such as a computer terminal window using the STR command modifier shown later.

3.1.10 Out of Range

When declaring variables, careful consideration should be given to the maximum value it will store. If a byte sized variable is declared but the result of some function is word sized then data will be lost. MCL has no way of knowing these possible conditions existing within your program.

3.1.11 Constants

Constants are defined values that will never change. The assigned value is stored in program memory and can not be changed. Constants are set during compile time by MCL. Constants are a convenient way to give a name to a numeric value. They are typically used to make a program more human readable. There are two types of constants. Normal constants defined using CON and an explicit floating point constant defined by FCON. You can use almost any name for a constant, except reserved words (see manual appendix).

Example:

```
LimitTravel CON 255
MaxSpeed FCON 6.50
```

3.1.12 Constant Tables

Tables are similar to arrays except tables only store constant values. You can store large strings of text in a constant table. Each position is accessed like arrays using an index number. Values assigned to a table are stored in program memory and can not be changed during program run time. A common use for constant tables is building interactive menu systems. If the user selects *n* reply with a string from table *n*. Constant tables are restricted to word boundaries if you have an odd number of bytes in a byte table an extra byte is added for padding automatically.

Example:

```
Sentence ByteTable "Hello World!"
```

The above example bytetable *Sentence* contains the string *Hello World!*. Each character including the space is a byte in the table. *Sentence* contains 12 bytes. To access the bytes you would use an index value of 0 to 11.

Example:

```
Temp VAR Byte
Sentence ByteTable "Hello World!"

Temp = Sentence(0)

The variable Temp now is equal to the ASCII value H since index 0 is the first byte in the
defined table.
```

3.1.13 Table Types

Type	Description
ByteTable	Each table index point is byte sized (8 bits).
SByteTable	Each table index point is sbyte sized(signed 8 bits).
WordTable	Each table index point is word sized (16 bits).
SWordTable	Each table index point is sword sized(signed 16 bits).
LongTable	Each table index point is long sized (32 bits).
SLongTable	Each table index point is slong sized (32 bits).
FloatTable	Each table index point is floating point sized (32 bits).

3.1.14 System Registers

System registers are special variables defined by MCL to access functions of the MCP control unit. System registers can, in most cases be used just like a user defined variable. However some System Registers are read only(see table below) and **Variable Modifiers** can not be used with system registers.

3.1.15 System Registers Table

Variable Names	Variable Size	Array Size	Type	Description
VERSION	WORD	1	R	Firmware Version
SYSSTATUS	WORD	1	R	Error and Warning Status
SYSCLK	LONG	1	R	System Clock Counter
SYSUSTICK	LONG	1	R	System Microsecond Tick
SYSTICK	LONG	1	R	System Millisecond Tick
SYSTEMP	WORD	1	R	System Temperature 1
SYSTEMP2	WORD	1	R	System Temperature 2(on select units)
SYSMBAT	WORD	1	R	System Main Battery Voltage
SYSLBAT	WORD	1	R	System Logic Battery voltage
SYSMINMBAY	WORD	1	R/W	Minimum Main Battery Voltage
SYSMAXMBAT	WORD	1	R/W	Maximum Main Battery Voltage
SYSMINLBAT	WORD	1	R/W	Minimum Logic Battery Voltage
SYSMAXLBAT	WORD	1	R/W	Maximum Logic Battery Voltage
DOUTACTION	BYTE	8	R/W	Action to trigger an Output Pin
DOUT	BIT	8	R/W	Output Pin State
PRIORITYLEVEL	BYTE	1	R	Control Priority Level Number
PRIORITYACTIVE	BYTE	1	R	Control Priority Level Mask
MOTORFLAGS	WORD	2	R	Motor Control State Flags
MOTORPWM	SWORD	2	R	Motor PWM Setting
MOTORCURRENT	SWORD	2	R	Motor Current Reading
MOTORTARGETPWM	SWORD	2	R	Motor Target PWM Setting
MOTORVELKP	LONG	2	R/W	Motor Velocity KP Value
MOTORVELKI	LONG	2	R/W	Motor Velocity KI Value
MOTORVELKD	LONG	2	R/W	Motor Velocity KD Value
MOTORVELQPPS	LONG	2	R/W	Motor Velocity QPPS Value
MOTORTARGETSPEED	SLONG	2	R	Motor Target Speed
MOTORDISTANCE	LONG	2	R	Motor Distance
MOTORPOSKP	LONG	2	R/W	Motor Position KP Value
MOTORPOSKI	LONG	2	R/W	Motor Position KI Value
MOTORPOSKIMAX	LONG	2	R/W	Motor Position KIMAX Value
MOTORPOSKD	LONG	2	R/W	Motor Position KD Value
MOTORPOSMAX	SLONG	2	R/W	Motor Maximum Position
MOTORPOSMIN	SLONG	2	R/W	Motor Minimum Position
MOTORPOSDEADZONE	LONG	2	R/W	Motor Position Deadzone
MOTORTARGETPOS	SLONG	2	R	Motor Target Position
MOTORACCEL	LONG	2	R	Motor Acceleration
MOTORDECCEL	LONG	2	R	Motor Decceleration

Variable Names	Variable Size	Array Size	Type	Description
MOTORSPEED	SLONG	2	R	Motor Speed
MOTORDEFAULTACCEL	LONG	2	R/W	Motor Default Acceleration
MOTORMAXCURRENT	SWORD	2	R/W	Motor Maximum Current
MOTORMINCURRENT	SWORD	2	R/W	Motor Minimum Current
MOTORL	LONG	2	R/W	Motor Inductance
MOTORR	LONG	2	R/W	Motor Resistance
MOTORENCPOS	SLONG	2	R/W	Encoder Position
MOTORENCSPEED	SLONG	2	R	Encoder Speed
MOTORENCSPEEDS	SLONG	2	R	Average Encoder Speed(1 second running average)
MOTORENCSTATUS	BYTE	2	R	Encoder Status
MOTORENCPIN	BYTE	2	R/W	Encoder Signal Input Pin
MOTORBUFFER	BYTE	2	R	Motor Command Buffer Position
STREAMTYPE	BYTE	4	R/W	Stream Type
STREAMRATE	LONG	4	R/W	Stream Rate
STREAMTIMEOUT	LONG	4	R/W	Stream Timeout
STREAMTICK	LONG	4	R	Stream Tick(System tick of last recevied byte)
STREAMISBUSY	BIT	4	R	Stream Is Busy
STREAMCOUNT	BYTE	4	R	Buffered bytes in stream
SIGNALACTIVE	BYTE	32	R	Signal Is Active
SIGNALTYPE	BYTE	32	R/W	Signal Type
SIGNALTARGET	BYTE	32	R/W	Signal Target
SIGNALMINACTION	BYTE	32	R/W	Minimum Signal Action
SIGNALMAXACTION	BYTE	32	R/W	Maximum Signal Action
SIGNALLOWPAS	BYTE	32	R/W	Low Pass Filter Setting
SINGALTIMEOUT	LONG	32	R/W	Signal Timeout
SIGNALLOADHOME	SLONG	32	R/W	Signal Loadhome Setting
SIGNALMIN	SLONG	32	R/W	Minimum Signal
SIGNALMAX	SLONG	32	R/W	Maximum Signal
SIGNALCENTER	SLONG	32	R/W	Center Signal
SIGNALDEADBAND	LONG	32	R/W	Signal Deadband
SIGNALPOWEREXP	SLONG	32	R/W	Signal Power Exponent
SIGNALPOWERMIN	LONG	32	R/W	Signal Power Minimum
SIGNALMODE	BYTE	32	R	Signal Mode
SIGNALMINOUT	SLONG	32	R/W	Signal Minimum Output
SIGNALMAXOUT	SLONG	32	R/W	Signal Maximum Output
SIGNALPOSITION	SLONG	32	R	Signal Position
SIGNALPERCENT	WORD	32	R	Signal Percent
SIGNALSPEED	SLONG	32	R	Signal Speed
SIGNALSPEEDS	SLONG	32	R	Average Signal Speed
SIGNALCOMMAND	SLONG	32	R	Signal Command

VERSION
The version variable holds the firmware version loaded on the board. This allows a MCL script to determine if the firmware version supports the functions being used by the script. The high byte of the word value stores the major version number and the low byte of the word value stores the minor verison number.

```
main
      puts 0,["Version ",hex2 VERSION.byte1\2,".", hex2 VERSION.byte0\2,13]
      pause 100
      goto main
```

SYSSTATUS
Variable holds the current controller warning and error states. It can be used in an MCL script to determine if an error has occured.

Status Bits	
Motor1 Overcurrent Protection	0x0001
Motor2 Overcurrent Protection	0x0002
ESTOP Triggered	0x0004
Temperature Error	0x0008
Temperature2 Error	0x0010
Main Battery High Error	0x0020
Main Battery Low Error	0x0040
Logic Battery High Error	0x0080
Logic Battery Low Error	0x0100
Main Battery High Warning	0x0800
Temperature Warning	0x4000

```
main
    puts 0,[hex4 SYSSTATUS\4,13]
    pause 100
    goto main
```

SYSCLK
variable holds the current clock counter value. The clock counter starts counting from 0 on power up in single clock increments((160mhz).

SYSUSTICK
variable holds the current tick counter value. The tick counter starts counting from 0 on power up in 1us increments.

SYSTICK
variable holds the current tick counter value. The tick counter starts counting from 0 on power up in 1ms increments.

SYSTEMP
variable holds the board temperature in 10ths of a degree Celcius increments.

```
main
    puts 0,[real TOFLOAT SYSTEMP/10.0\2,13]
    pause 100
    goto main
```

SYSTEMP2
Variable holds the boards secondary temperature in 10ths of a degree increments. Only MCP21xx support this feature. There is two temperature sensors one for each channel.

SYSMBAT
variable hols the main battery voltage reading in 10ths of a volt.

```
main
    puts 0,[real TOFLOAT SYSMBAT/10.0\2,13]
    pause 100
    goto main
```

SYSLBAT
Variable to read the logic battery voltage in 10ths of a volt. If the logic battery input is not being used the reading may float but will usually read as 0v. To prevent the Logic battery input from floating the user can ground the Logic Battery input.

```
main
    puts 0,[real TOFLOAT SYSLBAT/10.0\2,13]
    pause 100
    goto main
```

SYSMINMBAT
Variable to read or set the minimum main battery voltage limit. Values are in 10ths of a volt.

```
SYSMINMBAT = 120    ;tenths of a volt
main
    puts 0,[hex4 SYSSTATUS\4,13]    ;watch status for an error
    pause 100
    goto main
```

SYSMAXMBAT
Variable can be use to read or set the maximum main battery voltage limit. Values are in 10ths of a volt.

```
SYSMAXMBAT = 200    ;tenths of a volt
main
    puts 0,[hex4 SYSSTATUS\4,13]    ;watch status for a warning
    pause 100
    goto main
```

SYSMINLBAT
Variable can be use to read or set the minimum logic battery voltage limit. Values are in 10ths of a volt.

```
SYSMINLBAT = 120    ;tenths of a volt
main
    puts 0,[hex4 SYSSTATUS\4,13]    ;watch status for an error
    pause 100
    goto main
```

SYSMAXLBAT
Variable can be use to read or set the maximum logic battery voltage limit. Values are in 10ths of a volt.

```
SYSMAXLBAT = 120    ;tenths of a volt
main
    puts 0,[hex4 SYSSTATUS\4,13]    ;watch status for an error
    pause 100
    goto main
```

DOUTACTION(8)
Variable array is used to set or read the DOUT pins action setting.

Actions	
Motor1 is active	0x01
Motor2 is active	0x02
Either motor is active	0x03
Motor1 is reversed	0x04
Motor2 is reversed	0x05
Either motor is reversed	0x06
Overvoltage	0x07
Overtemperature	0x08
Stat1 LED	0x09
Stat2 LED	0x0A
Err LED	0x0B

```
;Output Error Status on an external LEDs/Lamps
DOUTACTION(0) = 0x0B      ;wire DOUT1 to an external LED/Lamp
DOUTACTION(1) = 0x07      ;wire DOUT2 to an external load dump circuit
```

DOUT(8)
Variable array is used to set or read the output state of a DOUT pin.

```
;Blink an external LED
main
      DOUT(0) = 1  ;wire DOUT1 to an external LED/Lamp
      pause 100
      DOUT(0) = 0  ;wire DOUT1 to an external LED/Lamp
      pause 100
      goto main
```

PRIORITYLEVEL
Variable holds the control priority level that is currently active. Levels 0,1 or 2.

```
main
      if PRIORIYLEVEL=0 then
            puts 0,["Prioritylevel 1",13]
      elseif PRIORITYLEVEL=1
            puts 0,["Prioritylevel 2",13]
      elseif PRIORITYLEVEL=2
            puts 0,["Prioritylevel 3",13]
      goto main
```

PRIORITYACTIVE
Variable holds the trigger mask for priority levels. The mask indicates which control types can become active. For exampel if Priority Level 1 is set to Serial and Oriority Level 2 is active, PRIORITYACTIVE will equal 0x01. Indicating any activity on a Serial stream will change the priority level back to level 1.

Priority Levels	
SERIAL	0x01
PULSE	0x02
ANALOG	0x04

MOTORFLAGS (2)
Variable array holds the control state flags for both motor channels. These can be useful when debugging hardware problems or triggering some specific script function based on motor status.

Flags	
stop triggered	0x0001
reverse triggered	0x0002
forward stop triggered	0x0004
backward stop triggered	0x0008
loadhome triggered	0x0010
estop triggered	0x0020
motor is active	0x0040
velocity pid is enabled	0x0080
position pid is enabled	0x0100
motor command buffere is empty	0x0200
distance command is active	0x0400

```
main
      if MOTORFLAGS(0)&0x0040 then
            puts 0,["Motor1 is active.",13]
      endif
      if MOTORFLAGS(1)&0x0040 then
            puts 0,["Motor2 is active.",13]
      endif
      pause 100
      goto main
```

MOTORPWM (2)
Variable array holds the current PWM setting for both motor channels.

```
main
	puts 0,["Motor 1 PWM:",dec MOTORPWM(0),13]
	pause 100
	goto main
```

MOTORCURRENT (2)
Variable array holds the current reading for both motor channels. Value is in 10ma increments

```
main
	puts 0,["Motor1 current:",real MOTORCURRENT(0)/100.0\2,13]
	pause 100
	goto main
```

MOTORTARGETPWM (2)
Variable array holds the target PWM setting for both motor channels. This is the PWM value the motor is increasing/decreasing power to reach.

```
main
	puts 0,["Motor1 pwm target:",real MOTORTARGETOWM(0)/100.0\2,13]
	pause 100
	goto main
```

MOTORVELKP (2)
Variable array holds the velocity Kp setting for both motor channels.

```
;initialize  motor Kp setting in script
MOTORVELKP(0) = TOINT(1.0*65536.0)
main
	goto main
```

MOTORVELKI (2)
Variable array holds the velocity Ki setting for both motor channels.

```
;initialize  motor Ki setting in script
MOTORVELKI(0) = TOINT(0.5*65536.0)
main
	goto main
```

MOTORVELKD (2)
Variable array holds the velocity Kd setting for both motor channels.

```
;initialize  motor Kd setting in script
MOTORVELKD(0) = TOINT(0.25*65536.0)
main
	goto main
```

MOTORVELQPPS (2)
Variable array holds the QPPS setting for both motor channels.

```
;initialize  motor QPPS setting in script
MOTORVELQPPS(0) = 180000
main
       goto main
```

MOTORTARGETSPEED (2)
Variable array holds the target Speed setting for both motor channels. This is the speed the motor is accelerating/deccelerating to reach.

```
main
       puts 0,["Motor1 target speed:",dec MOTORTARGETSPEED(0),13]
       pause 100
       goto main
```

MOTORDISTANCE (2)
Variable array holds the remaining distance for both motor channels.

```
main
       puts 0,["Motor1 distance:",dec MOTORDISTANCE(0),13]
       pause 100
       goto main
```

MOTORPOSKP (2)
Variable array holds the position Kp setting for both motor channels.

```
;initialize  motor Kp setting in script
MOTORPOSKP(0) = TOINT(4000.0*2048.0)
main
       goto main
```

MOTORPOSKI (2)
Variable array holds the position Ki setting for both motor channels.

```
;initialize  motor Ki setting in script
MOTORPOSKI(0) = TOINT(0.0*2048.0)
main
       goto main
```

MOTORPOSKIMAX (2)
Variable array holds the position KiMax setting for both motor channels.

```
;initialize  motor KiMax setting in script
MOTORPOSKIMAX(0) = 100
main
       goto main
```

MOTORPOSKD (2)
Variable array holds the position Kd setting for both motor channels.

```
;initialize  motor Kd setting in script
MOTORPOSKD(0) = TOINT(40000.0*2048.0)
main
       goto main
```

MOTORPOSMAX (2)
Variable array holds the position Max setting for both motor channels.

```
;initialize  motor maximum position
MOTORPOSMAX(0) = 10000
main
       goto main
```

MOTORPOSMIN (2)
Variable array holds the position Min setting for both motor channels.

```
;initialize  motor minimum position
MOTORPOSMIN(0) = 10000
main
       goto main
```

MOTORPOSDEADZONE (2)
Variable array holds the position Deadzone setting for both motor channels.

```
;initialize  motor deadzone
MOTORDEADZONE(0) = 10
main
       goto main
```

MOTORTARGETPOS (2)
Variable array holds the target Position for both motor channels. This is the current position to move to.

```
main
       puts 0,["Motor1 Target Position:",sdec MOTORTARGETPOS(0),13]
       pause 100
       goto main
```

MOTORACCEL (2)
Variable array holds the current acceleeration setting for both motor channels.

```
main
       puts 0,["Motor1 Acceleration Setting:",dec MOTORACCEL(0),13]
       pause 100
       goto main
```

MOTORDECCEL (2)
Variable array holds the current decceleration setting for both motor channels. The seperate decceleration value is only used with position commands. Acceleration is used as both acceleration and decceleration in velocity/distance commands.

```
main
       puts 0,["Motor1 Decceleration Setting:",dec MOTORDECCEL(0),13]
       pause 100
       goto main
```

MOTORSPEED (2)
Variable array holds the current speed setting for both motor channels.

```
main
       puts 0,["Motor1 speed:",sdec MOTORSPEED(0),13]
       pause 100
       goto main
```

MOTORDEFAULTACCEL (2)
Variable array holds the default acceleration setting used for duty cycle commands and compatibility commands for both motor channels.

```
;initialize  motor default acceleration for Duty cycle commands
MOTORDEFAULTACCEL(0) = 655360
main
       goto main
```

MOTORMAXCURRENT (2)
Variable array holds the maximum current setting for both motor channels.

```
;initialize  motor maximum current limit
MOTORMAXCURRENT(0) = 3000
main
       goto main
```

MOTORMINCURRENT (2)
Variable array holds the minimum current setting for both motor channels.

```
;initialize  motor maximum current limit
MOTORMINCURRENT(0) = -3000
main
       goto main
```

MOTORL (2)
Variable array holds the motor inductance setting for both motor channels. If this value and MOTORR are set then PWM commands are executed using a current control algorithm instead of voltage control.

```
main
      puts 0,["Motor1 inductance:",real TOFLOAT MOTORL(0)/16777216.0\2,13]
      pause 100
      goto main
```

MOTORR (2)
Variable array holds the motor resistance setting for both motor channels. If this value and MOTORL are set then PWM commands are executed using a current control algorithm instead of voltage control.

```
main
      puts 0,["Motor1 resistor:",real TOFLOAT MOTORR/16777216.0\2,13]
      pause 100
      goto main
```

MOTORENCPOS (2)
Variable array holds the current encoder position for both motor channels. This value is equal to the signal channel assigned as encoder input for the specific motor.

```
main
      puts 0,["Motor1 encoder pos:",sdec MOTORENCPOS,13]
      pause 100
      goto main
```

MOTORENCSPEED (2)
Variable array holds the current encoder speed for both motor channels. This value is equal to the signal channel assigned as encoder input for the specific motor.

```
main
      puts 0,["Motor1 encoder speed:",sdec MOTORENCSPEED,13]
      pause 100
      goto main
```

MOTORENCSPEEDS (2)
Variable array holds the current encoder average speed for both motor channels. This value is equal to the signal channel assigned as encoder input for the specific motor.

```
main
      puts 0,["Motor1 encoder average speed:",sdec MOTORENCSPEEDS,13]
      pause 100
      goto main
```

MOTORENCSTATUS (2)
Variable array holds the current encoder status for both motor channels. This value is equal to the signal channel assigned as encoder input for the specific motor.

```
main
      puts 0,["Motor1 status:",hex MOTORENCSTATUS,13]
      pause 100
      goto main
```

MOTORENCPIN (2)
Variable array holds the signal number used for encoder input. This allows any signal pin(P0,P1,PID1A,PID1B etc) to be used as an encoder input. Pulse, PWM, analog or quadrature inputs can be used as encoder input.

```
main
      puts 0,["Motor1 encoder pin:",dec MOTORENCPIN(0),13]
      pause 100
      goto main
```

MOTORBUFFER (2)
Variable array holds the motor buffer state for both motor channels. The high bit of the byte indicates the motor buffer is empty and no buffered command is executing. The low 7 bits indicate the used buffer size(0 to 127).

```
main
      puts 0,["Motor1 buffer status:",hex2 MOTORBUFFER(0)\2,13]
      pause 100
      goto main
```

STREAMTYPE (4)
Variable array holds the stream type for each stream. Stream types currently supported are 0(user stream) or 1(packetserial stream).

```
;initialize  USB stream type
STREAMTYPE(0) = 0
main
      goto main
```

STREAMRATE (4)
Variable array holds the transmission speed for each stream. This value is in bits per second(bps).

```
;initialize RS232 stream rate
STREAMRATE(3) = 115200
main
      goto main
```

STREAMTIMEOUT (4)
Variable array holds the activity timeout in milliseconds for each stream. The timeout value is the maximum time between receiving data on the channel before the channel is deactivated and a low priority channel is activated.

```
;initialize  USB stream timeout to 1 second
STREAMTIMEOUT(0) = 1000
main
      goto main
```

STREAMTICK (4)
Variable array holds the last activity tick count for each stream. This is the last system tick count when data was received on the specified stream.

```
main
      puts 0,["USB Stream last active tick:",dec STREAMTICK(0),13]
      pause 100
      goto main
```

STREAMISBUSY (4)
Variable array holds the busy status flag for each stream. This indicates if the stream is active or not.

```
main
      puts 0,["USB Stream state:",dec STREAMISBUSY(0),13]
      pause 100
      goto main
```

STREAMCOUNT (4)
Variable array holds the used byte count for each stream buffer.

```
main
      puts 0,["USB Stream bytes waiting:",dec STREAMCOUNT(0),13]
      pause 100
      goto main
```

SIGNALACTIVE (32)
Variable array holds the activity status for each signal. This indicates if the signal input is active

```
main
      puts 0,["P0 state:",dec SIGNALACTIVE(0),13]
      pause 100
      goto main
```

SIGNALTYPE (32)
Variable array holds the signal type for each signal.

Signal Types	
None	0x00
Analog	0x01
RC Pulse/PWM	0x02
Quadrature Encoder	0x03

```
main
      puts 0,["P0 SignalType:",dec SIGNALTYPE(0),13]
      pause 100
      goto main
```

SIGNALTARGET (32)
Variable array holds the target for each signal. The target can be either Motor1(0) or Motor2(1).

```
main
      puts 0,["P0 Signal Target:",dec SIGNALTARGET(0),13]
      pause 100
      goto main
```

SIGNALLOWPASS (32)
Variable array holds the low pass filter setting. 2^n samples. 0 = disabled.

SIGNALTIMEOUT (32)
Variable array holds the activity timeout for each signal. The time in system tick counts before the signal is deactivated and a lower priority input will be activated.

```
main
      puts 0,["P0 Signal Timeout:",dec SIGNALTIMEOUT(0),13]
      pause 100
      goto main
```

SIGNALTICK (32)
Variable array holds the last activity tick count for each signal. The time the last signal was received.

```
main
      puts 0,["P0 Last Signal Tick:",dec SIGNALTICK(0),13]
      pause 100
      goto main
```

SIGNALMINACTION (32)
Variable array holds the action that will trigger if the minimum is reached for each signal.

Action triggered by Signal Minimum	
Motor1 safestop	0x0001
Motor2 safestop	0x0100
Both Motors safestop	0x0101
Motor1 off	0x0002
Motor2 off	0x0200
Both Motors off	0x0202
Motor1 reverse	0x0004
Motor2 reverse	0x0400
Both Motors reverse	0x0404
Motor1 Forward Limit	0x0008
Motor2 Forward Limit	0x0800
Both Motors Forward Limit	0x0808
Motor1 Reverse Limit	0x0010
Motor2 Reverse Limit	0x1000
Both Motors Reverse Limit	0x1010
Motor1 load Loadhome value	0x0020
Motor2 load loadhome value	0x2000
Both Motors load loadhome	0x2020
Run Script	0x0040
Stop Script	0x4000
Reset Script	0x0080
Trigger ESTOP	0x8000

```
;initialize  Signal P0 minimum action
SIGNALMINACTION(0) = 0x8000      ;trigger E-Stop if signal goes low
main
       goto main
```

SIGNALMAXACTION (32)
Variable array holds the maximum action for each signal. See SIGNALMINACTION for actions list.

```
;initialize  P0 maximum action
SIGNALMAXACTION(0) = 0x0040  ;run script if P0 goes high
main
       goto main
```

SIGNALLOADHOME (32)
Variable array holds the loadhome value for each signal. This value is loaded to the encoder position value when a Loadhome action occurs. This only applies to encremental encoder signals.

```
main
       puts 0,["P0 Load Home Value:",dec SIGNALLOADHOME(0),13]
       pause 100
       goto main
```

SIGNALMIN (32)
Variable array holds the minimum position for each signal. This limits the minimum signal value and can trigger an action(on a DOUT or a motor channel).

```
main
      puts 0,["P0 Minimum:",dec SIGNALMIN(0),13]
      pause 100
      goto main
```

SIGNALMAX (32)
Variable array holds the maximum position for each signal. This limits the maximum signal value and can trugger an action(on a DOUT or a motor channel).

```
main
      puts 0,["P0 Maximum:",dec SIGNALMAX(0),13]
      pause 100
      goto main
```

SIGNALCENTER (32)
Variable array holds the center position for each signal. This value sets the center point of the signal input. This in combination with the min and max values can be used to limit the maximum range of either direction of a signal.

```
main
      puts 0,["P0 Center:",dec SIGNALCENTER(0),13]
      pause 100
      goto main
```

SIGNALDEADBAND (32)
Variable array holds the deadband width for each signal. The deadband allows you to elliminate any slop in your control input when it is centered. For example you use an analog joystick that doesnt return to center perfectly you can set an area near the center to always equal the center value.

```
main
      puts 0,["P0 Deadband:",dec SIGNALDEADBAND(0),13]
      pause 100
      goto main
```

SIGNALPOWEREXP (32)
Variable array holds the power exponent for each signal. The power exponent value lets the user define the curve of the signal input. This allows the control to be more precise near the center point or near the min and max limits. The higher/lower the number the more extreme the change in control.

```
main
      puts 0,["P0 Exp:",real SIGNALPOWEREXP/65536.0\2,13]
      pause 100
      goto main
```

SIGNALPOWERMIN (32)
Variable array holds the minimum power level for each signal. This sets the absolute minimum control output for a signal controling a motor.

```
main
      puts 0,["P0 Power Minimum:",dec SIGNALPOWERMIN,13]
      pause 100
      goto main
```

SIGNALMODE (32)
Variable array holds the mode for each signal. Changign the signal mode allows the user to read specific dat from a signal:

Mode	
Absolute position of the signal	0x00
Percent of the signal(PWM Duty)	0x01
Average speed of the signal Change	0x02
Speed of the signal change	0x03

```
main
      puts 0,["P0 Mode:",dec SIGNALMODE(0),13]
      pause 100
      goto main
```

SIGNALMINOUT (32)
Variable array holds the minimum output for each signal. Sets the minimum output to the target motor.

```
main
      puts 0,["P0 Minimum Output:",sdec SIGNALMINOUT(0),13]
      pause 100
      goto main
```

SIGNALMAXOUT (32)
Variable array holds the maximum output for each signal. Sets the maximum output to the target motor.

```
main
      puts 0,["P0 Maximum Output:",dec SIGNALMAXOUT(0),13]
      pause 100
      goto main
```

SIGNALPOSITION (32)
Variable array holds the current position for each signal.

```
main
      puts 0,["P0 Position:",dec SIGNALPOSITION(0),13]
      pause 100
      goto main
```

SIGNALPERCENT (32)
Variable array holds the current percentage of the signal throw for each signal.

```
main
      puts 0,["P0 PERCENT:",dec SIGNALPERCENT(0),13]
      pause 100
      goto main
```

SIGNALSPEED (32)
Variable array holds the current speed for each signal.

```
main
      puts 0,["P0 SPEED:",dec SIGNALSPEED(0),13]
      pause 100
      goto main
```

SIGNALSPEEDS (32)
Variable array holds the current average speed for each signal.

```
main
      puts 0,["P0 Average Speed:",dec SIGNALSPEEDS(0),13]
      pause 100
      goto main
```

SIGNALCOMMAND (32)
Variable array holds the command for each signal. This is either the position,percent,speed or average speed of the signal input depending on the signal mode setting.

Command Types	
Position	0
Percent	1
Speed	2
Average Speed	3

```
main
      puts 0,["P0 Output Command:",dec SIGNALCOMMAND(0),13]
      pause 100
      goto main
```

3.2 MCL Math

3.2.1 Math Functions
MCL includes a full complement of math and comparison functions. MCL supports 32 bit integer math, both signed and unsigned. It also supports floating point math. A signed value denotes whether the resulting value is positive or negative.

3.2.2 Number Bases
Although all calculations are handled internally in binary, users can refer to numbers as decimal, hexadecimal or binary, whichever is most convenient for the programmer. For example, the number 2349 can be referred to as:

2349	Decimal
0x092D	Hexadecimal
%100100101101	Binary

Leading zeros are not required for hex or binary numbers, but may be used if desired. When using signed integers (sbyte, sword, slong) it is probably a good idea to stick to decimal notation to avoid confusion.

3.2.3 Math and Operators
Operators are what makes math work, by performing a function. An example of an operator would be + (Addition), - (Subtraction), * (Multiplication) and / (Division). All these symbols represent an operation to be performed. However, the operators need something to do, so we add operands which are better known as arguments. Math arguments are the values used in an expression.

In the following section you will see the word “expression” used many times. This refers to something like 1+2. The expression 1+2 has one operator (+) and two arguments(1 and 2) or operands.

3.2.4 Operators

Operator	Description
-	Changes the value of an expression from positive to negative. Also used in subtraction.
ABS	Returns the absolute value of an expression.
SIN	Returns the integer sine of an expression.
COS	Returns the integer cosine of an expression.
SQR	Returns the integer square root.
BIN2BCD	Converts expression from binary to packed binary coded decimal format.
RANDOM	Returns a random 32 bit number generated from a seed value.
-	Subtraction. Also used to sign a value.
+	Addition.
*	Multiplication, returns the low 32bits of a integer multiplication result.
/	Division.
**	Returns high 32 bits of an integer multiplication result.
*/	Fractional integer multiplication.
//	Remainder of Integer Division.

Operator	Description
MAX	Limits the expression to a maximum value
MIN	Limits the expression to a minimum value
DIG	Returns the specified digits value
<<	Shift left by specified amount.
>>	Shift right by specified amount.
&	Binary math AND.
|	Binary math OR.
^	Binary math XOR.
&/	Binary math NAND.
|/	Binary math NOR.
^/	Binary math NXOR.
=	Is equal to.
<>	Is not equal to.
<	Is less than.
>	Is greater than.
<=	Is less than or equal to.
>=	Is greater than or equal to.
AND	Logical AND.
OR	Logical OR.
XOR	Logical XOR.
NOT	Logical NOT.
TOINT	Convert a Floating Point value to a Integer
TOFLOAT	Convert an Integer value to a Floating Point
FABS	Floating Point Absolue value
FSQRT	Floating Point Square Root
FSIN	Floating Point Sine
FCOS	Floating Point Cosine
FTAN	Floating Point Tangent
FASIN	Floating Point ArcSine
FACOS	Floating Point ArcCosine
FATAN	Floating Point Arc Tangent
FSINH	Floating Point Hyperbolic Sine
FCOSH	Floating Point Hyperbolic Cosine
FTANH	Floating Point Hyperbolic Tangent
FATANH	Floating Point Hyperbolic ArcTangent
FLN	Floating Point Natural Log
FEXP	Floating Point Exponent

3.2.5 Operator Precedence

All math functions have a precedence order. This simply means each math function in an expression is calculated based on its precedence not necessarily based on the order in which it appears in the expression. This even holds true for math as it was taught in school. However, the precedence of order may differ.

To solve the following equation 2+2*5/10 = 3 MCL would start with multiplication first since it has the higher precedence order. 2*5 will be calculated first which equals 10, then divide 10 by 10 which equals 1, then addition of 2 which equals 3. The 2 was added last since it had the lowest precedence.

The multiplication and division operators have equal precedence, and both have higher precedence than addition and subtraction. Now you can change the order in which the math is performed by using parenthesis. This will force a specific order. Using parentheses, the following expression ((2+2)*5) /10 would yield a result of 2.

3.2.6 Precedence Table

Order	Operation
1st	NOT, ABS, SIN, COS, - (NEG), SQR, RANDOM, TOINT, TOFLOAT, BIN2BCD, ~(Binary NOT), !(Binary NOT), NOT(Logical NOT), FABS, FSQRT, FSIN, FCOS, FTAN, FASIN, FACOS, FATAN, FSINH, FCOSH, FTANH, FATANH, FLN, FEXP
2nd	Dig
3rd	MAX, MIN
4th	*, **, */, /, //
5th	+, -
6th	<<, >>
7th	<, <=, =, >=, >, <>
8th	&, |, ^, &/, |/, ^/
9th	And, Or, Xor

- (Negative)

Signs an expression (integer or floating point) as a negative value.

```
Temp var Byte
Result var Byte
Temp = 1
Result = Temp + -1
```

Temp is first is set to equal 1. Then -1 is added. -1 is a signed integer. So Result now equals 0. Since 1 added to -1 equals 0.

ABS
The Absolute Value (ABS) converts a signed number to its absolute value. The absolute value of a number is the value that represents its difference from 0. The absolute value of -4 is 4. If the number is positive the result will always be the same number returned:

```
temp = abs(-1234)
temp = abs(1234)
```

In the example above, the result will always be 1234 since the difference of 0 from -1234 is 1234.

SIN, COS
In integer arithmetic, some modifications to the way sine and cosine work have been made. For example, in floating point math, the expression:

```
ans = sin(angle)
```

where angle is 1 radian, would return a value of 0.841... for ans. In fact, the sine of an angle must always be a fractional value between -1 and 1. MBasic can not deal with fractional values for integer math so SIN and COS are made to work with integers.

Since we are dealing with binary integers, we divide the circle into 256 (rather than 360) parts. This means that a right angle is expressed as 64 units, rather than 90 degrees. When working with Basic Micro Studio angular units give you a precision of about 1.4 degrees.

The result of the SIN or COS function is a signed number in the range of –127 to +128. This number divided by 128 gives the fractional value of SIN or COS.

In most “real world” applications, the angle does not need to be in degrees, nor does the result need to be in decimal form. The following example shows a possible use of SIN values. If a sensor returns the angle of a robotic control arm as a number from 0 to 64, where 0 is parallel and 64 is a right angle. We want to take action based on the sine of the angle.

```
        limit var byte
        angle var byte
loop
        (code that inputs the value of “angle”)
        limit = sin angle
        if limit > 24 then first
        if limit > 48 then second
        goto loop
first
        code to warn of excessive angle
        goto loop
second
        code to shut down equipment
        etc...
```

This will warn the operator if the arm angle exceeds approximately 8 units (11.25 degrees) and shut down the equipment if the arm angle exceeds approximately 16 units (22.5 degrees). Most control examples don't need to work in actual degrees or decimal values of sine or cosine. To find the sine of a 60 degree angle, first convert the angle to MBasic units by multiplying by 256 and dividing by 360. For example:

```
angle = 60 * 256 / 360
```

will result in a value of 42. (It should actually be 42.667, which rounds to 43, but with integer arithmetic the decimal fraction is ignored, and the integer is not rounded up.) Then find the sine of this angle:

```
ans = sin angle
```

This will give a result of 109. Dividing this value by 128 will give the decimal value of 0.851 (compared to the correct floating point value which should be 0.866). You can not directly get the decimal value by doing this division within MBasic (you would get a result of 0). However, you could first multiply by 1000, then divide by 128 to get 851 as your result.

SQR (Square Root)
SQR returns the integer portion of the square root of the argument. Increased precision can be obtained by multiplying the argument by an even square of 10, such as 100 or 10000.

If the value of "num" is 64, the following statement will return the value of 8 (which is the square root of 64).

```
answer = sqr num
```

If the value of "num" is 220, the following statement will return the value 14, which is the integer portion of 14.832..., the square root of 220.

```
answer = sqr num
```

If more precision is required, multiply the argument by 100 or 10000. Using the example where "num" = 220 a value 148 is returned, which is 10 times the square root of 220.

```
answer = sqr (num * 100)
```

Alternately,

```
answer = sqr (num * 10000)
```

the above example will return the value 1483, which is 100 times the square root of 220.

BIN2BCD
This command let you convert binary to "packed" binary coded decimal (BCD). A BCD number is one in which each decimal digit is represented by a 4 bit binary number (from 0 to 9). Packed BCD packs two 4 bit decimal digits in a single byte of memory.

For example, the decimal number 93 is represented in binary as:

Values	128	64	32	16	8	4	2	1
Binairy	0	1	0	1	1	1	0	1

The same number is expressed in packed BCD as:

Values	8	4	2	1	8	4	2	1
Binairy	1	0	0	1	0	0	1	1

Assuming that "answer" is a byte variable and "num" has the decimal value of 93, the statement

```
answer = bin2bcd num
```

will set answer to a binary value of 10010011 (which is 93 in packed BCD).

RANDOM
Random generates a 32 bit (Long) random number from the seed value. As with most random number generators, the random numbers generated will follow a predictable pattern depending on the seed value(ie: it is psuedo random), and each time the program is run the random number sequence will be the same if the seed value is the same. The code snippet below will return a pseudo random set of numbers by re-seeding from the results:

```
seed var long

seed = 123456
seed = random seed
```

There are steps that can be taken to avoid repeating random number sequences. This is typically done using hardware based features. One common method is using an internal hardware timer for the seed value and asking the user to press a button at the beginning of a game. Each time the button is pressed the timer value will likely be different. Another method is reading an A/D pin that is left floating and near a noisy signal trace.

Subtraction (-)
Subtract a value (integer or floating point) from another value. The resulting number is not signed unless a signed variable is used. An example of subtraction:

```
time var byte
time = 100
time = time - 1
```

The variable time will now equal 99 since we subtracted 1 from 100.

Addition (+)
Add one value (integer or floating point) to another value. The resulting number is not signed unless a signed variable is used. An example of addition:

```
time var byte
time = 100
time = time + 1
```

The variable time will now equal 101 since we added 1 to 100.

Multiplication (*)
Multiply one value (integer or floating point) by another value. The resulting number is not signed unless a signed variable is used. An example of multiplication:

```
time var byte
time = 100
time = time * 1
```

The variable time will now equal 100 since we multiplied 100 by 1.

Division (/)
Divide one value (integer or floating point) by another value. Integer division discards fractional results. For example:

```
result = 76/7
```

will set the variable “result” to a value of 10. (The actual decimal result would be 10.857, but the decimal part is discarded, rounding is not done.) If your application requires fractional results you can use floating point numbers or the following solution.

Use a floating point variable instead to get the full precision.

```
result var float
result = 76.0/7.0
```

Alternatively, when using integer variables multiply the dividend by 10, 100, 1000 etc. before dividing. The result will gain extra digits of precision but must be interpreted correctly. Using the previous example we can gain three digits of precision as follows. This is known as fixed point division:

```
temp = dividend * 1000     ;dividend is now 76000
result = temp/7
```

The example sets "result" to a value of 10857.

High Multiplication (**)
If two long variables or constants are multiplied, the result may exceed 32 bits. Normally, the multiply function will return the least significant (lowest) 32 bits. The ** function will, instead, return the most significant 32 bits.

```
time = 80000 ** 80000 ; result returns high 32 bits
```

The value of time would be equal to 0x1 which is the high 32 bits of the result 6,400,000,000.

Fractional Multiplication (*/)
Fractional multiplication will multiply a number with a fractional part. The multiplier must be a long value which is handled by a special method. The high 16 bits are the integer portion of the multiplier, the low 16 bits are the fractional part (expressed as a fraction of 65536). The result will be an integer with any fractional remainder discarded (not rounded).

Let us say we want to multiply the number 346 x 2.5. The multiplier must be constructed as follows. The high 16 bits will have a value of 2. We can do this with:

```
mult.highword = 2
```

The low 16 bits will have a value that is half of 65535 or 32782 so:

```
mult.lowword = 32782
```

Then we do the fractional multiply:

```
a = 346 */ mult
```

The example will give "a" the value 865. A similar procedure will let you multiply by any fraction by expressing that fraction with a denominator of 65535 as closely as possible.

Notice that half of 65535 is actually 32782.5; a number we can not enter as the fractional part. This means that multiplication by exactly half is not possible. However, the difference is so small that it has no effect on the actual outcome of the integer result.

Mod (//)
The mod function (short for "modulo") returns the remainder after an integer division. So, for example, 13 modulo 5 is 3 (the remainder after dividing 13 by 5).

The mod function can be used to determine if a number is odd or even, as shown here:

```
x var word
      y var word
      (code that sets the value of x)
      y = x//2
      if y=0 goto even    ;zero indicates an even number
      if y=1 goto odd     ;one indicates an odd number
even
      (more code)
odd
      (more code)
```

Similarly, the mod function can be used to determine if a number is divisible by any other number.

MAX
The MAX function returns the smaller of two expressions (integer or floating point). For example:

```
x var word
y var word
code to set value of y
x = y max 13
```

The example above will set x to the value of y or 13, whichever is smaller. Think of this as x equals y up to a maximum value of 13.

MIN
The MIN function returns the larger of two expressions (integer or floating point). For example:

```
x var word
y var word
code to set value of y
x = y min 9
```

The example will set y to the value of x or 9, whichever is larger. Think of this as x equals y down to a minimum value of 9.

DIG
The DIG (digit) function is used to isolate a single digit of a decimal number. For example:

```
x var word
y var byte
(code to set y)     ;say the result is y=17458
x = y dig 4  ;gives the 4th digit of y, which is 7
```

Digits are counted from the right, starting with 1. The DIG function will work with numbers in decimal format only. If you need to find a specific digit in a hex or binary number, use a variable modifier.

Shift Left (<<)
The Shift Left operator shifts all the bits of a value to the left by a specified amount. Shifting left is the same as multiplying the value by 2 to the *n*th power. Bits shifted off the left end are lost. Zeros are added to the right for vacant bits. The example program will display the value of time before and after shifting left by 4. The results will be displayed in binary:

```
time var byte

serout sout, i9600,[bin time]
time = time << 4
serout sout, i9600,[bin time]
```

Important: The sign bit is not preserved so this function should not be used with signed numbers.

Shift Right (>>)
The Shift Right operator shifts all the bits of a value to the right by a specified amount. Shifting right is the same as dividing the value by 2 to the *n*th power. Bits shifted off the right end are lost. Zeros are added to the left for vacant bits. The example program will display the value of time before and after shifting right by 4. The results will be displayed in binary:

```
time var byte

serout sout, i9600,[bin time]
time = time >> 4
serout sout, i9600,[bin time]
```

Important: The sign bit is not preserved so this function should not be used with signed numbers.

AND (&)
The AND (&) function is a binary operator. It sets the result to 1 if both bits are 1's or 0 if either or both bits are 0's.

1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	0	0	0	1	0	0	0	1

One useful function for AND is to "mask" certain bits of a number. For example, if we are interested only in the low 4 bits of a number, and wanted to ignore the high 4 bits, we could AND (&) the number with 00001111 as shown here:

Value1	0	1	0	1	1	1	0	1
Value2	0	0	0	0	1	1	1	1
Result	0	0	0	0	1	1	0	1

As you can see, the high 4 bits are now all set to 0's, regardless of their original state, but the low 4 bits retain their original state.

OR (|)
The OR (|) function is a binary operator. It sets the bit to 1 if either or both of the matching bits are 1 or to 0 if both bits are 0's.

1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	1	1	0	1	1	1	1	1

Exclusive OR (^)
The ^ function is a binary operator. It sets the resulting bits to a 1 if either, but not both, of the matching bits are 1 or to 0 otherwise.

1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	1	1	0	0	1	1	1	0

NAND (&/)

The NOT AND function is a binary operator. It compares the bits of two values. It sets the result bits to 1 if neither bit is set or to a 1 for all other cases.

1 &/ 1 = 1
1 &/ 0 = 1
0 &/ 1 = 1
0 &/ 0 = 0

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	1	1	1	0	1	1	1	0

NOR (|/)
The NOT OR function is a binary operator. It compares two values bit by bit and sets the result to a 1 if neither bit is a 1, all other conditions will return a 0.

1 |/ 1 = 0
1 |/ 0 = 0
0 |/ 1 = 0
0 |/ 0 = 1

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	0	0	1	0	0	0	0	0

NXOR (^/)
The NOT XOR function is a binary operator, It compares two values bit by bit and sets the result to a 1 if neither bit is a 0 or both bits are a 1. All other conditions will return a 0.

1 ^/ 1 = 1
1 ^/ 0 = 0
0 ^/ 1 = 0
0 ^/ 0 = 1

Value1	0	1	0	1	1	1	0	1
Value2	1	0	0	1	0	0	1	1
Result	0	0	1	1	0	0	0	1

Equal (=)
The Equal (=) is a logic operator and also is used to set a variable to some value.

```
if(temp = 10)then
endif

or

temp = 10
```

The two examples compares the value of temp to 10 and sets temp to 10 using "=".

NOT Equal To (<>)
The NOT Equal (<>) is a logic operator and compares to see if a value is not equal to another value.

```
if temp <> 10 then
```

The conditional statement will check to see if temp is not equal to 10. If the value of temp is lower or greater the comparison will be true.

Less Than (<)
The Less Than (<) is a logic operator and compares to see if a value is less than another value.

```
if temp < 10 then
```

The conditional statement will check to see if temp is less than 10. If the value of temp is lower the comparison is true. Therefore, any value equal to or over 10 will be false.

Greater Than (>)
The Greater Than (>) is a logic operator and compares to see if a value is greater than another value.

```
if temp > 10 then
```

The conditional statement will check to see if temp is greater than 10. If the value of temp is higher, the comparison is true. Therefore, any value from 0 to 10 will be false (Less Than or Equal To).

The Less Than or Equal To (<=) is a logic operator and compares if something is less than or equal to some value.

```
if temp <= 10 then
```

The conditional statement will check to see if temp is less than or equal to 10. If the value of temp is less than 10 or equal to 10 the comparison is true. Therefore, any value from 11 and up is false.

Greater Than or Equal To (>=)
The Greater Than or Equal To (>=) is a logic operator and compares to see if a value is greater than or equal to another value.

```
if temp >= 10 then
```

The conditional statement will check to see if temp is greater than or equal to 10. If the value of temp is greater than 10 or equal to 10 the comparison is true. Therefore, any value from 0 to 9 is false.

AND
The AND operator is a logic comparison operator. It compares two conditions to make a single true or false statement. The AND operator will return a true only if both conditions are true. If one condition is false then a false is returned. The truth table demonstrates all combinations:

Condition 1	Condition 2	Result
True	True	True
True	False	False
False	True	False
False	False	False

The AND operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. It differs from the & operator which is used in binary math functions. Example of the AND operator:

```
if minute = 10 AND hour = 1 then alarm
```

The conditional statement will check to see if both expressions are true before returning a true and jumping to the alarm label. If one of the expressions is not true, a false is returned and the label is skipped. The IF..THEN only jumps to the label if the statement is true.

OR
The OR operator is a logic comparison operator. It compares two conditions to make a single true or false statement. The OR operator will return a true if one or both conditions are true. If both conditions are false then a false is returned. The truth table belwo demonstrates all combinations:

Condition 1	Condition 2	Result
True	True	True
True	False	True
False	True	True
False	False	False

The OR operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. It differs from the | operator which is used in binary math functions. Example of the OR operator:

```
if hour = 12 OR minute = 30 then ding
```

The conditional statement will check to see if either expression is true before returning a true and jumping to the ding label. If both of the expressions are false the label is skipped. The IF.. THEN only jumps to the label if the statement is true.

XOR
The XOR operator is a logic comparison operator. It compares two conditions to make a single true or false statement. The XOR operator will return a true if one but not both conditions are true. If both conditions are true or false then a false is returned. The truth table below demonstrates all combinations:

Condition 1	Condition 2	Result
True	True	False
True	False	True
False	True	True
False	False	False

The XOR operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. It differs from the ^ operator which is used in binary math functions. Example of the XOR operator:

```
if hour > 5 XOR hour = 5  then QuitTime
```

The conditional statement will check to see if either expression is true before returning a true and jumping to the quittime label. If both of the expressions are false or true the label is skipped. The IF..THEN only jumps to the label if the statement is true.

NOT
The NOT operator is a logic operator that inverts a condition. When used, true becomes false and false becomes true. The truth table demonstrates all combinations:

Condition	Result
True	False
False	True

The NOT operator is used in decision making commands such as IF..THEN, DO..WHILE and so on. Example of the NOT operator:

```
if hour = 5 then Quit_Time
if NOT hour < 5 then Over_Time
```

If hour is not equal to 5 the first conditional statement will skip Quit_Time. In the second conditional statement if hour is not less than 5 it jumps to the label Over_Time since the NOT operator inverted the result of the condition. The IF..THEN only jumps to the label if the statement is true.

Floating Point Operators

Operator	Description
TOINT	Convert a Floating Point value to a Integer
TOFLOAT	Convert an Integer value to a Floating Point
FSQRT	Floating Point Square Root
FSIN	Floating Point Sine
FCOS	Floating Point Cosine
FTAN	Floating Point Tangent
FASIN	Floating Point ArcSine
FACOS	Floating Point ArcCosine
FATAN	Floating Point Arc Tangent
FSINH	Floating Point Hyperbolic Sine
FCOSH	Floating Point Hyperbolic Cosine
FTANH	Floating Point Hyperbolic Tangent
FATANH	Floating Point Hyperbolic ArcTangent
FLN	Floating Point Natural Log
FEXP	Floating Point Exponent

TOINT
The TOINT operator explicitly converts a floating point value into an integer value. The decimal point of the floating point number is truncated.

```
myfloat var float
myint var long

myfloat = 10.0 ;myfloat now equals 10.0
myfloat = myfloat * 123.123 ;myfloat now equals 1231.23
myint = TOINT myfloat ;myint now equals 1231
serout s_out,i9600,["result = ",sdec myint,13]
```

TOFLOAT
The TOFLOAT operator explicitly converts an integer value into a floating point value.

```
myint var long
myfloat var float

myint = 10
myfloat = TOFLOAT myint / 100.0 ;myfloat now equals 0.1
serout s_out,i9600,["result = ",real myfloat,13]
```

FABS
The Absolute Value (ABS) converts a signed number to its absolute value. The absolute value of a number is the value that represents its difference from 0. The absolute value of -4 is 4. If the number is positive the result will always be the same number returned:

```
temp = fabs(-1234.1234)
temp = fabs(1234.1234)
```

In the example above, the result will always be 1234.1234.

FSQRT
FSQRT returns the floating point square root of the argument. A square root is the number, that when multiplied by itself will equal the original argument.

If the value of "num" is 2, the following statement will return the value of 1.4142:

```
myfloat var float

myfloat = FSQRT 2.0
serout s_out,i9600,["result = ",real myfloat,13]
```

FSIN
The FSIN operator calculates the floating point Sine of an angle in radians. FSIN gives the ratio of the length of the side opposite of the angle to the length of the hypotenuse in a right triangle.

```
myangle var float
mysin var float

myangle = 3.14159/2.0 ;myangle now equals PI/2 degrees in radians
mysin = FSIN myangle ;mysin now equals 1.0
serout s_out,i9600,["mysin = ",real mysin,13]
```

FCOS
The FCOS operator calculates the floating point cosine of an angle in radians. FCOS gives the ratio of the length of the side adjacent the angle, to the length of the hypotenuse in a right triangle.

```
myangle var float
mycos var float

myangle = 0.0 ;myangle now equals 0 degrees in radians
mycos = FCOS myangle ;mycos now equals 1.0
serout s_out,i9600,["mycos = ",real mycos,13]
```

FTAN
The FTAN operator calculates the floating point tangent of an angle in radians. FTAN gives the ratio of the length of the side opposite the angle to the length of the side adjacent to the angle in a right triangle

```
myangle fcon 3.14159/4 ;myangle is 45 degrees in radians
myadj fcon 100.0
myopp var float

;This calcualtion finds the triangles opposite side length
;for a right triangle with angle of 45 degrees and
;adjacent side of 100.
myopp = FTAN myangle * myadj ;myopp now equals 100.0
serout s_out,i9600,["myopp = ",real myopp,13]
```

FASIN
The FASIN operator calculates the floating point arc sine of a value. FASIN returns the angle in radians given the ratio of the length of the side opposite the angle and the length of the hypotenuse in a right triangle.

```
mysin fcon 1.0
myangle var float

myangle = FASIN mysin ;myfloat equals PI/2
serout s_out,i9600,["myangle = ",real myangle,13]
```

FACOS
The FACOS operator calculates the floating point arc cosine of a value. FACOS returns the angle in radians given the ratio of the length of the side adjacent the angle and the length of the hypotenuse in a right triangle.

```
mycos fcon 0.0
myangle var float

myangle = FACOS mycos ;myfloat equals 0
serout s_out,i9600,["myangle = ",real myangle,13]
```

FATAN
The FATAN operator calculates the floating point arc tangent of a value. FATAN returns the angle in radians given the ratio of the length of the side opposite the angle and the length of the side adjacent to the angle in a right triangle.

```
mytan fcon 1.0
myangle var float

myangle = FATAN mytan ; myfloat equals PI/4
serout s_out,i9600,["myangle = ",real myangle,13]
```

FLN
The FLN operator calculates the floating point natural log of a value. The natural log is used to calculate the time it takes for compound growth to reach the specified amount. For example FLN 20.08 will equal approximately 3. This means it takes 3 growth cycles (the amount of time it takes to grow 100%) to reach 20.08 times the original amount.

```
result var float

result = FLN 2.0 ;result now equals 0.69315
serout s_out,i9600,["result = ",real result,13]
```

FEXP
The FEXP operator calculates the Floating Point Natural Exponent of a value. The natural exponent calculates the inverse of the natural log. Given time how much will something grow. FEXP 3 will equal approximately 20.08 times the original quantity.

```
result var float

result = FEXP 0.693115 ;result now equals 2.0
serout s_out,i9600,["result = ",real result,13]
```

FSINH
The FSINH operator calculates the floating point hyperbolic Sine of the value.

```
param var float
result var float

param = FLN 2.0
result = FSINH param
serout s_out,i9600,["The hyperbolic sine of ",real param," is ",real result,13]
```

FCOSH
The FCOSH operator calculates the floating point hyperbolic cosine of the value.

```
param var float
result var float

param = FLN 2.0
result = FCOSH param
serout s_out,i9600,["The hyperbolic cosine of ",real param," is ",real result,13]
```

FTANH
The FTANH operator calculates the floating point hyperbolic tangent of the value.

```
param var float
result var float

param = FLN 2.0
result = FTANH param
serout s_out,i9600,["The hyperbolic tangent of ",real param," is ",real result",13]
```

FATANH
The FATANH operator calculates the floating point hyperbolic arc tangent of the value.

```
param var float
result var float

param = 0.6
result = FEXP (FATANH param)
serout s_out,i9600,["The result is ",real result,13]
```

3.3 MCL Modifiers

3.3.1 Output Modifiers

Modifiers can be use when outputing data using the PUTS command or when storing data in a variable using the LET(=) command.

An example of a command modifier is formatting a decimal value into a printable string. The decimal value 32 would output to a terminal window as a space character(ASCII code 32 is a space). Instead, to display the actual decimal value of the variable in a human readable format you would use the DEC modifier:

```
Temp Var Byte
Temp = 32

puts 0,[DEC TEMP] ;send the ascii characters "32" to the USB port
```

This code snippet above will send the ascii characters "3" and "2" to the USB port. If the DEC modifier was not used, a space character (" "), binary value 32, would be sent instead

Modifiers can be used to modify values stored in arrays with the LET(=) commands:

```
string var byte(100)

string = "Hello World"     ;string(0-10)="Hello World"
string = dec 1234567       ;string(0-6)="1234567"
string = ihex 0x3456       ;string(0-4)="$3456"
```

3.2.2 Modifiers

Name	Description
DEC	Decimal
SDEC	Signed Decimal
HEX	Hexadecimal
IHEX	Indicated ($) Hexadecimal
REP	Repeat character *n* times
REAL	Floating point number with decimal point
STR	Write specified number of characters(can be used to copy arrays)

DEC
DEC{#max} expression{\#min}

#max: optional maximum number of digits to send
#min: optional minimum number of digits to send

The DEC modifier when used in an output command will convert a stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 1234.

```
temp var word
temp = 1234

puts 0, [DEC temp] ;prints "1234"
```

```
DEC(#max) variable
```

#max: optional maximum number of digits to receive

The DEC modifier for input commands will format incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent decimal numbers up to 9 characters long and store the converted value in a variable. Until a numeral is received (e.g. "0" to "9") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 9 characters have been received.

```
temp var word

puts 0, [DEC temp]
```

SDEC
SDEC{#max} expression{\#min}

#max: optional maximum number of digits to send
#min: optional minimum number of digits to send

The SDEC modifier when used in an output command will convert a stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 1234.

```
temp var sword
temp = -1234

puts 0, [SDEC temp] ;prints "-1234"
```

```
SDEC(#max) variable
```

#max: optional maximum number of digits to receive

The SDEC modifier for input commands will format incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent decimal numbers up to 9 characters long and store the converted value in a variable. Until a negative sign or numeral is received (e.g. "0" to "9") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 9 characters have been received.

```
temp var sword

puts 0, [SDEC temp]
```

HEX
HEX{#max} expression{\#min}

#max: optional maximum number of digits to send
#min: optional minimum number of digits to send

The HEX modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 12ab.

```
temp var word
temp = 0x12ab

puts 0, [HEX temp] ;prints "0x12ab"
```

```
HEX(#max) variable
```

#max: optional maximum number of digits to receive

The HEX modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexidecimal numbers up to 8 characters long and store the converted value in a variable. Until a numeral is received (e.g. "0" to "9" or "a" to "f") any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var word

puts 0, [HEX temp]
```

IHEX
IHEX{#max} expression{\#min}

#max: optional maximum number of digits to send
#min: optional minimum number of digits to send

The IHEX modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display $12ab.

```
temp var word
temp = 0x12ab

puts 0, [SHEX temp] ;prints "$12ab"

IHEX(#max) variable
```

#max: optional maximum number of digits to receive

The IHEX modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent hexidecimal numbers up to 8 characters long and store the converted value in a variable. Until the indicator ($) is received, any incoming characters are ignored. Once the indicator has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var word

puts 0, [IHEX temp]
```

REP
The REP modifier will output the character *n* a specified number of times. The example will repeat the specified character "A" 10 times.

```
puts 0, [REP "A"\10] ;prints A 10 times
```

REAL
REAL{#maxint} expression{\#maxdec}

#maxint: optional maximum number of integer digits to send
#maxdec: optional maximum number of decimal point digits to send

The REAL modifier, in output commands, converts stored values to ASCII characters. The example will format the value of *temp* so it prints out the number in a terminal window. The output would display 3.14159.

```
temp var float
temp = 3.14159

puts 0, [REAL temp] ;prints "3.14159"

REAL variable
```

The REAL modifier, for input commands, formats incoming ASCII characters into a numeric value. The example will read in ASCII characters that represent floating point numbers and store the converted value in a variable. Until a numeral is received any incoming characters are ignored. Once a numeral has been received any character not a numeral will cause the conversion to finish before 8 characters have been received.

```
temp var float

puts 0, [REAL temp]
```

STR
STR array\length{\eol}

The STR modifier will output *Length* amount of characters from specified constant or variable array until the end of the array or until an optional specified *EOL* character is found.

```
temp var byte(20)
temp = "Hello world",0

puts 0,[str temp\20\0] ;output "Hello world"

STR array\length{\eol}
```

The STR modifier, in input commands, will receive *Length* number of characters and store them in the variable array specified. An optional end of line (*EOL*) character can be specified to tell STR to stop receiving characters.

```
temp var byte(20)

puts 0,[str temp\20\13] ; receive upto 20 characters
```

3.4 MCL Commands

3.4.1 Command Reference

The syntax and example is provided for each command in the command reference section of this manual. In addition, Ion Studio MCL editor has built in syntax help. As a command is typed, The editor will show the syntax for that command as its being typed.

BRANCH

Syntax
branch index, [label1, label2, label3, ..., labelN]

- **Index** - is a variable, constant or expression that is used to reference a label in a list. The index is 0 based.

- **Label -** is a list of constant labels that are jump points in a program.

Description
The BRANCH command will jump to a label defined within the brackets. The label used for the jump is determined by the pointer *index*. The only limits to the number of labels within the brackets is the available program memory.

Example
Connect to the following program with the terminal window at 9600 baud. Enter a value from 0 to 4. The program will jump to the label specified by the value typed in. The BRANCH command is a great way to build a long conditional jump list based on some value range. User interactive menu systems are one possibility.

```
Index var word

Main
       Pause 1000
       for Index = 0 to 4
              gosub jumptable
       next
       Goto Main

jumptable
       branch Index, [label0, label1, label2, label3, label4]

Label0
       puts 0, [13, "Label 0"]
       return
Label1
       puts 0, [13, "Label 1"]
       return
Label2
       puts 0, [13, "Label 2"]
       return
Label3
       puts 0, [13, "Label 3"]
       return
Label4
       puts 0, [13, "Label 4"]
       returnCLEAR
```

CLEAR

Syntax
clear

Description
The clear command sets all user memory to zero. The CLEAR function is typically used in the beginning of a program to set all memory to a know state. In some situations CLEAR is used in place of several statements like variable = 0.

DIST

Syntax
dist motor,speed,accel,distance,buffer

- **Motor** - is a variable, constant or expression that specifies which motor to access.

- **Speed -** is a variable, constant or expression that sets the speed to run the motor at.

- **Accel -** is a variable, constant or expression that sets the acceleration/decceleration to use when the motor speeds up or slows down to the specified speed.

- **Distance -** is a variable, constant or expression that specifies the distance to move the motor.

- **Buffer -** is a variable, constant or expression that determines if the command will be buffered(0) or execute immediately(1) overriding any currently running command on the motor.

Description
The DIST command is used to control a motor using an encoder with a Velocity PID control algorithm. The user must have tuned the Velocity PID settings for the motor before using the DIST command. With a properly tuned Velocity PID accurate relative movments can be made by the motor. Note that multiple DIST commands can be chained together by buffering the commands. As each command finishes the next will start executing.

```
;code assumes the velcoity PID settings for the specific motor are correctly tuned.
main
      dist 0,12000,12000,12000,0
      dist 0,-12000,12000,12000,0

      ;wait until second command has started to execute and repeat
      while(MOTORBUFFER(0)&0x7F<>0)
      wend

      goto main
```

DIST2

Syntax
dist speed1,accel1,distance1,speed2,accel2,distance2,buffer

- **Speed1 & Speed2 -** is a variable, constant or expression that sets the speed to run the motor at.
- **Accel1 & Accel2 -** is a variable, constant or expression that sets the acceleration/decceleration to use when the motor speeds up or slows down to the specified speed.
- **Distance1 & Distance2 -** is a variable, constant or expression that specifies the distance to move the motor.
- **Buffer -** is a variable, constant or expression that determines if the command will be buffered(0) or execute immediately(1) overriding any currently running command on the motor.

Description
The DIST2 command is used to control both motors using encoders with a Velocity PID control algorithm. The user must have tuned the Velocity PID settings for both motors before using the DIST2 command. With a properly tuned Velocity PID accurate relative movments can be made by the motor. Note that multiple DIST2 commands can be chained together by buffering the commands. As each command finishes the next will start executing.

```
;code assumes the velcoity PID settings for the motors are correctly tuned.
main
       dist2 12000,12000,12000,-12000,12000,12000,0
       dist2 -12000,12000,12000,12000,12000,21000,0

       ;wait until second command has started to execute and repeat
       while(MOTORBUFFER(0)&0x7F<>0)
       wend

       goto main
```

DO - WHILE

Syntax
do program statements while condition

- **Statements -** any group of commands to be run inside the loop.

- **Condition** - can be a variable or expression

Description
The DO - WHILE loop executes commands nested inside of it while some condition is true. The condition can be any variable or expression that is tested every loop until it is false. Zero (0) is false, not zero (0) is true. By default DO - WHILE will test this condition. The loop will continue until its value equals zero (0). DO - WHILE will always run at least once since the condition is checked at the end of the loop. Multiple DO - WHILE commands can be nested within each other. However you can not nest DO - WHILE with a WHILE - WEND together.

```
temp var long

main
	temp = 0;
	do
		puts 0,[dec temp,13]
		temp = temp + 1
		pause 100
	while(temp<10);
	goto main
```

END

Syntax
end

Description
END stops program execution until the unit is reset.

FOR...NEXT

Syntax
for countVal = startVal to finishVal {step increment}
...code...
next
- **CountVal** - a variable used to store the current count.

- **StartVal** - the starting value to count from.

- **FinishVal** - a value to count up or down to.

- **Increment** - the value to increment the variable by through each loop.

Description
Repeats a block of instructions and increments a variable counter with a value specified each time through the loop. The FOR...NEXT loop will exit when the count value is no longer between start and finish. When no step increment is defined the default increment value is 1. If the finishVal is modified by the instructions in the FOR...NEXT block the loop can be forced to exit early. You can safely jump out of a FOR...NEXT loop.

Example
The example below will send the results to the USB port. It will count up from 0 to 9 then exit.

```
value var long
main
      for value = 0 to 9
      pause 1000
      puts 0,["Value=",dec value,13]
next
end
```

The example below will print the results to a terminal window at 9600 baud. It will count down from 9 to 0 then exit.

```
value var long
main
      for value = 9 to 0
      pause  1000
      puts 0,["Value=",dec value,13]
next
end
```

The example below will print the results to a terminal window at 9600 baud. It will count up from 0 to 54 using 5 as the increment value then exit.

```
value var long
main
      for value = 0 to 54 step 5
      pause 1000
      puts 0,["Value=",dec value,13]
next
end
```

GETS

Syntax
gets stream,[variable1,...,variableN]

- **Stream** - is a variable, constant or expression that specifies which stream to read.

- **Variabl1...VariablN -** is a list of variables to store the read values in.

Description
The GETS command is used to read bytes from a data stream and store them in user variables.

Stream Types

USB Virtual Communications Port	0
CAN (CANOpen or raw CAN)	1
TTL UART	2
RS232 UART	3

Example
The example reads 4 bytes from the USB stream and stores them in a long variable:

```
value var long

main
      gets 0,[value.byte3, value.byte2, value.byte1, value.byte0]
      puts 0,[dec value,13]
end
```

GOSUB

Syntax
gosub label{[argument1,...,argumentN]}{,DataResult}

- **Label** - the go to label of the subroutine to be executed.

- **Argument** - is user defined arguments to send to the called subroutine. The only limit to the amount of arguments used is program memory.

- **DataResult** - is an optional variable to store the value returned from called subroutine.

Description
The GOSUB command will jump to a specified label. After executing the code at the jump label a RETURN command is then used to return the program to the next command after the last called GOSUB.

There is no limit to this with Basic Micro Studio other than the size of the available stack memory. GOSUB stores the address of the next command on the stack and jumps to the specified label. User specified arguments can be defined in the subroutine. A return value from the subroutine can be stored in the variable that is specified by the GOSUB DataResult argument.

Notes
Subroutines must always exit via the RETURN command, which clears the saved address from the stack and returns to the command following the calling GOSUB.
User defined arguments must match the number of arguments defined at the subroutine. If they do not match, a stack overflow or underflow will happen.
If a subroutine returns a value, the GOSUB is not required to use it.

Example
The program below will print the results to the terminal window at 9600 baud. The results will be 110. The GOSUB command has two arguments and includes the DataResult variable. The values 10 and 100 are passed to the subroutine MyAdd. The values are then loaded into the variables arg1 and arg2. Since RETURN can have an expression the variables arg1 and arg2 are added and returned to the variable result.

```
Result var long

Main
      Gosub myadd[10,100],result
      puts 0,["Result =",dec result]
End

Arg1 var long
Arg2 var long

MyAdd [arg1,arg2]
Return arg1+arg2
```

See Also
RETURN

GOTO

Syntax
goto label
- **Label** - is a label the program will jump to.

Description
The GOTO command tells the program to jump to some label.

Examples
The following program is a simple loop using GOTO that will repeat forever.

```
basic
       Pause 800
       puts 0,[0, "Basic"]
       pause 800
       goto micro
       goto basic

micro
       puts 0,[2, " Micro"]
       pause 800
       Goto rules
       goto basic

rules
       puts 0,[2, " Rules!"]
       pause 800
       goto basic
```

IF...THEN...ELSEIF...ELSE...ENDIF

Simple Syntax
if expression then label
if expression then goto label
if expression then gosub label

Extended Syntax
if expression then
...code...
endif

if expression then
...code...
else
...code...
endif

if expression then
...code...
elseif expression
...code...
endif

if expression then
...code...
elseif expression
...code...
else
...code...
endif

Description
IF..THEN commands are the decision makers of MBasic. IF..THEN evaluates a condition to determine if it is true. If you want to test the value of a variable against another variable or a known value, you would use the IF..THEN commands. When the results are true, the code after THEN is executed. If it returns false, the code after THEN will be ignored. A simple example would be to increment a variable in a loop and each time through the loop test if the variable equals 10. This lets us control how many times through the loop we want to run. We can also test if our variable is greater than, less than or not equal too. Several math expressions can be used for the test condition.

```
main
if temp = 10 then label
goto main

label
goto main
```

The above statement is looking for the variable *temp* to equal 10. If the comparison is true then we will jump to *label*. If the comparison is not true, then lets keep looping. Since there is nothing to make the variable equal to 10 the code snippet would loop forever.

Notes
Multiple ELSEIF blocks may be used in a single IF...THEN block.
ELSE will only execute following code if no conditions were true.
ENDIF is required to close a block of conditionals.

Example
This first example demonstrates using the IF...THEN argument with a goto *label*. If something is true jump to the label after the THEN statement. Otherwise, if the condition is false execute the commands on the next line after the THEN statement. You can follow the program flow with a terminal window connected at 9600 baud.

```
value var long
value = 0

main
       value = value+1
       if value = 10 then reset

       ;display the value on the PC terminal window
       puts 0,["Value=",dec value,13]
       pause 1000
goto main

reset
       value = 0
goto main
```

GOSUB Example
A GOSUB statement can be used after a THEN command. When the condition is true a GOSUB will send the program to the GOSUB label. Eventually a RETURN statement is expected. This will return the program to the next line of code after the GOSUB statement was used. It is an easy way to create conditional branching in a main program loop. The program will increment value by 1 each loop. Once value is equal to 10 the condition becomes true and the GOSUB label is executed. This resets value to 0 starting the process over. The program was written to be followed using a terminal window connected to it at 9600 baud. Follow the results until you understand the decision making process. Can you guess what the terminal window will show?

```
value var long
value = 0

main
       value = value+1
       if value = 10 then gosub reset

       ;display the value on the PC terminal window
       puts 0,["Value=",dec value,13]
       pause 1000
goto main

reset
       value = 0
return
```

Advanced Arguments
Now that we have covered the basics of the IF..THEN commands we can explore optional syntax. The next section will explain each with sample code that will display the result so you can follow along.

ENDIF Example
In some cases you may want to run a block of code when the condition is true. MCL can execute a command or commands directly after a THEN statement, instead of jumping to a *label*. The ENDIF is used to tell MCL run the following commands after THEN if the condition is true. So ENDIF literately mean what it says, lets end the IF. When the IF..THEN condition is false ENDIF tells MBasic to instead, run the commands after ENDIF. This simply resumes normal program operation. ENDIF is an easy way to execute a group of commands based on some condition returning true or completely skipping them if the condition is false.

```
value var long
value = 0

main
       value = value+1
       if value = 10 then
              value = 0
       endif

       ;display the value on the PC terminal window
       puts 0,["Value=",dec value,13]
       pause 1000
goto main
```

ELSEIF Example
In some cases you may want to test for several possible conditions. ELSEIF allows you to check multiple possible conditions until one of them is true using a single IF..ENDIF block. Until one of the ELSEIF conditions is found to be true each condition will be tested. As soon as one is found to be true, the code in that section will execute and the program execution will continue starting after the ENDIF.

```
if temp = 10 then
       temp = 20
elseif temp = 20
       temp = 10
endif
```

There is no limit to how many ELSEIF statements you can have in any single IF...THEN block. The following program shows an example using multiple ELSEIF statements.

```
value var long
value = 0

main
	if value = 0 then
		value = 1

	elseif value = 1
		value = 2

	elseif value = 2
		value = 3

	elseif value = 3
		value = 4

	elseif value = 10
		value = 0
	endif

	;display the value on the PC terminal window
	puts 0,["Value=",dec value,13]
	pause 1000
goto main
```

ELSE

In previous examples we tested several conditions that were determined as being false, which would cause the program will resume normal operation, skipping any code found enclosed within the IF..THEN / ENDIF statements unless an ELSE statement was added. The ELSE block will execute when everything else was FALSE. The following program was written to be followed using a terminal window connected at 9600 baud. See if you can you guess the results?

```
value var long
value = 0

main
	if value = 1 then
	value = 1

	elseif value = 2
	value = 2

	else
	value = 3

endif

	;display the value on the PC terminal window
	serout s_out,i9600,[“Value=”,dec value,13]
	pause 1000
goto main
```

I2COUT..I2COUTNS

Syntax
i2cout {label,timeout}, [databyte1, ..., databyteN]
i2coutns {label,timeout}, [databyte1, ..., databyteN]

- **Label** - is a label to jump to if the timeout triggers.

- **Timeout** - is the time in milliseconds to wait for a byte to transfer.

- **DataBytes** - is a list of byte sized variables or expressions that are the values to be sent to an attached I2C device. The only limit to the number of data bytes in one command is determined by the device itself.

Description
The I2COUT command combined with the I2COUTNS and I2CIN command can communicate with any I2C compatible devices. The specific data to send and receive from any particular I2C device will depend on the device. In general you will send an address/control byte that lets the I2C device know you are trying to talk to it, since multiple devices can be on one bus, and in what direction you will be talking to it, sending or receiving. The difference between I2COUT and I2COUTNS is that I2COUTNS does not send a stop signal when it finishes. This allows the user to send a repeated start by executing another I2COUT/I2COUTNS orI2CIN command.

```
;example to read a value from an MPU-6050
temp var long

main
      pause 10

      ;send WHOAMI command
      i2cout error,10,[0xd0,0x75]

      ;read reply
      i2coutns error,10,[0xd1]
      i2cin error,10,[temp]

      puts 0,[dec temp,13]

      goto main

error
      puts 0,["I2C Error:",dec i2cerror,13]
      goto main
```

I2CIN

Syntax
i2cin {label, timeout}, [databyte1,...,databyteN]

- **Label** - is a label to jump to if the timeout triggers.

- **Timeout** - is the time in milliseconds to wait for a byte to transfer.

- **DataBytes** - is a list of byte sized variables that store the values read from an attached I2C device. The only limit to the number of data bytes in one command is determined by the device itself.

Description
The I2CIN command is used to received data back from an I2C device after an appropriate I2COUT or I2COUTNS command has been sent. The exact format of the data to be received will depend on the I2C device being used.

Example
See I2COUT

MOVE

Syntax
move motor,speed,accel,deccel,position,buffer

- **Motor** - is a variable, constant or expression that specifies which motor to access.

- **Speed -** is a variable, constant or expression that sets the speed to run the motor at.

- **Accel -** is a variable, constant or expression that sets the acceleration to use when the motor start its movement.
- **Deccel -** is a variable, constant or expression that sets the decceleration to use when the motor stops its movement.
- **Position -** is a variable, constant or expression that sets the position for the motor to move to.
- **Buffer -** is a variable, constant or expression that determines if the command will be buffered(0) or execute immediately(1) overriding any currently running command on the motor.

Description
The MOVE command is used to control a motor using an encoder with a Position PID control algorithm. The user must have tuned the Position PID settings for the motor before using the MOVE command. With a properly tuned Position PID accurate absolute movments can be made by the motor. Note that multiple Move commands can be chained together by buffering the commands. As each command finishes the next will start executing.

```
;example assumes velocity/position  PID settings are correct for the specified motor
main
       move 0,1000,0,0,1000,0
       pause 2000   ;move should take 1 second then stops for 1 second
       move 0,1000,0,0,-1000,0
       pause 2000   ;move should take 1 second then stops for 1 second
       goto main
```

MOVE2

Syntax
move2 speed1,accel1,deccel1,position1, speed2,accel2,deccel2,position2, buffer

- **Motor1...Motor2** - is a variable, constant or expression that specifies which motor to access.

- **Speed1...Speed2 -** is a variable, constant or expression that sets the speed to run the motor at.
- **Accel1...Accel2 -** is a variable, constant or expression that sets the acceleration to use when the motor start its movement.

- **Deccel1...Deccel2 -** is a variable, constant or expression that sets the decceleration to use when the motor stops its movement.

- **Position1...Position2 -** is a variable, constant or expression that sets the position for the motor to move to.

- **Buffer -** is a variable, constant or expression that determines if the command will be buffered(0) or execute immediately(1) overriding any currently running command on the motor.

Description
The MOVE2 command is used to control both motors using encoders with a Position PID control algorithm. The user must have tuned the Position PID settings for the motors before using the MOVE2 command. With a properly tuned Position PID accurate absolute movments can be made by the motors. Note that multiple Move2 commands can be chained together by buffering the commands. As each command finishes the next will start executing.

```
;example assumes velocity/position  PID settings are correct for the motors
main
      move2 1000,0,0,1000,1000,0,0,-1000,0
      pause 2000   ;move should take 1 second then stops for 1 second
      move2 1000,0,0,-1000,1000,0,0,1000,0
      pause 2000   ;move should take 1 second then stops for 1 second
      goto main
```

PAUSE

Syntax
pause time

- **Time** - is a variable, constant or expression that specifies the number of milliseconds to wait.

Description
The PAUSE command is used to create a predetermined delay in a program. The amount of delay is specified in milliseconds. There are 1000 milliseconds in 1 second. If a value of 500 is used for time, it would be a half second. If 1000 is used for time, the program would wait 1 second.

POWER

Syntax
power motor,duty,accel

- **Motor** - is a variable, constant or expression that specifies which motor to access.
- **Duty -** is a variable, constant or expression that sets the duty to run the motor at.
- **Accel -** is a variable, constant or expression that sets the acceleration/decceleration to use when the motor powers up/down to the specified power.

Description
The POWER command is used to control a motor using just a PWM duty cycle percentage. This is the simplest method of controlling a DC motor but does not compensate for changes in the load on the motor.

```
;Simple PWM control of motor
main
      power 0,32767,32767 ;should take 1 second to accel to specific power(100% duty)
      pause 1000
      power 0,0,32767     ;should take 1 second to deccel to 0% power.
      pause 1000
      goto main
```

POWER2

Syntax
power2 duty1,accel1,duty2,accel2

- **Duty1...Duty2 -** is a variable, constant or expression that sets the duty to run the motor at.

- **Accel1...Accel2 -** is a variable, constant or expression that sets the acceleration/decceleration to use when the motor powers up/down to the specified power.

Description
The POWER2 command is used to control both motors using just PWM duty cycle percentages. This is the simplest method of controlling a DC motor but does not compensate for changes in the load on the motors.

```
;Simple PWM control of motor
main
      power2 32767,32767,0,32676er(100% duty)
      pause 1000
      power2 0,32767,32767,32767
      pause 1000
      goto main
```

PUTS

Syntax
puts stream,[{modifiers}data1,...,{modifiers}dataN]

- **Stream** - is a variable, constant or expression that specifies the stream to send data to.

- **Modifiers -** see the Modifiers section of the manual for specific syntax and usage.

- **Data1...DataN -** are variables, constants or expressions that specifies the data to be sent to the specified stream.

Description
The PUTS command is used to send data to an active stream, for communciations with with microcontrollers, computers or networks(see CAN). The PUTS command is also usefull when debugging your MCL script.

Stream Types

USB Virtual Communications Port	0
CAN (CANOpen or raw CAN)	1
TTL UART	2
RS232 UART	3

READ

Syntax
read address, databyte

- **Address** - is a byte sized variable or constant that specifies what address to read the on-board EEPROM from.

- **DataByte** - is a byte sized variable (0-255) which stores the data returned from the on-board EEPROM.

Description
All modules except the BasicATOM Pro 24 and BasicATOMPro ONE come with built in EEPROM. The READ / WRITE commands were created to access the built-in memory. READ will read a single byte from the built in EEPROM at the address specified in the command.

Example
The example program will write the string "Hello" starting at the first location to the on-board EEPROM. Next, it will read the EEPROM locations 0 to 10 and print the contents to the terminal window.

```
index var byte
char var byte

write 0,"H"
write 1,"e"
write 2,"l"
write 3,"l"
write 4,"o"

for index = 0 to 10
       read index,char
       puts 0,[char]
next
end
```

REPEAT...UNTIL

Syntax
repeat
program statements
until condition

- **Statements -** any group of commands to be run inside the loop.

- **Condition** - can be a variable or expression

Description
The REPEAT - UNTIL loop executes commands nested inside of it until some condition is false. This is the opposite of DO - WHILE and WHILE - WEND. The condition can be any variable or expression and is tested every loop until true.

Notes
The loop will continue until it's value is NOT equal to 0. REPEAT - UNTIL checks the condition at the end of the loop. This means the loop will always execute atleast once. You can nest multiple REPEAT - UNTIL commands within each other. You can nest DO - WHILE or WHILE - WEND loops within a REPEAT - UNTIL loop.

Example
The program will start counting up from 0 to 100. Once the index reaches a value of 101, the condition is no longer false. The greater than symbol (>) was used for the condition and 101 is now greater than 100 making the condition true. Since REPEAT - UNTIL loops while a statement is false the program will now exit.

```
;ALL - all_repeat_until.bas

Index var word


Main
       Index = 0

       Repeat
       index = index + 1

       ;lets print the value index
       puts 0,[0, "Couting: ", dec index]
       pause 75

       Until index > 100 ;run until index is greater than 100

       puts 0, [13,13, "Index = ", dec index]
       puts 0, [13, "My condition is no longer false."]
       puts 0, [13, "Index is now greater than 100"]
End
```

RETURN

Syntax
return {DataResult}

- **DataResult** - an optional value to return to the gosub statement, that can be an expression, constant or variable.

Description
GOSUB stores the address of the next command on the stack and jumps to the specified label. After executing the code at the jump label a RETURN command is used to remove, from the stack, the address stored by GOSUB and then jumps to that address.

Notes
Subroutines must exit via the RETURN command, which clears the saved address from the stack and returns to the command following the GOSUB. User defined arguments must match the number of arguments defined at the subroutine. If they do not match, a stack overflow or underflow will happen. If a subroutine returns a value, the GOSUB is not required to use it or specify a return value variable.

Example
The result will be 110. The GOSUB command has two arguments and includes a DataResult variable. The values 10 and 100 are passed to the subroutine MyAdd. The values are then loaded into the variables *arg1* and *arg2*. RETURN can have an expression, so the variables *arg1* and *arg2* are added and returned to the variable *result*.

```
Result var long

Main
      Gosub myadd[10,100],result
      puts 0,["Result =",dec result]
End

Arg1 var long
Arg2 var long

MyAdd [arg1,arg2]
Return arg1+arg2
```

See Also:
GOSUB

SPEED

Syntax
speed motor,speed,accel

- **Motor** - is a variable, constant or expression that specifies which motor to access.

- **Speed -** is a variable, constant or expression that sets the speed to run the motor at.

- **Accel -** is a variable, constant or expression that sets the acceleration/decceleration to use when the motor speeds up or slows down to the specified speed.

- **Buffer -** is a variable, constant or expression that determines if the command will be buffered(0) or execute immediately(1) overriding any currently running command on the motor.

Description
The SPEED command is used to control a motor using an encoder with a Velocity PID control algorithm. The user must have tuned the Velocity PID settings for the motor before using the SPEED command. With a properly tuned Velocity PID accurate speed control can be acheived by the motor. Note that multiple SPEED commands can be chained together by buffering the commands. As each command finishes the next will start executing.

```
;example assumes velocity PID settings are correct set.
main
       speed 0,12000,12000
       pause 1000
       speed 0,0,12000
       pause 1000
       speed 0,-12000,12000
       pause 1000
       speed 0,0,12000
       pause 1000
       goto main
```

SPEED2

Syntax
speed2 speed1,accel1, speed2,accel2

- **Speed1...Speed2 -** is a variable, constant or expression that sets the speed to run the motor at.

- **Accel1...Accel2 -** is a variable, constant or expression that sets the acceleration/decceleration to use when the motor speeds up or slows down to the specified speed.

- **Buffer -** is a variable, constant or expression that determines if the command will be buffered(0) or execute immediately(1) overriding any currently running command on the motor.

Description
The SPEED2 command is used to control both motors using encoders with a Velocity PID control algorithm. The user must have tuned the Velocity PID settings for the motors before using the SPEED2 command. With a properly tuned Velocity PID accurate speed control can be acheived by the motors. Note that multiple SPEED2 commands can be chained together by buffering the commands. As each command finishes the next will start executing.

```
;example assumes velocity PID settings are correct set.
main
       speed2 12000,12000,0,12000
       pause 1000
       speed2 0,12000,12000,12000
       pause 1000
       speed2 -12000,12000,0,12000
       pause 1000
       speed2 0,12000,-12000,12000
       pause 1000
       goto main
```

STOP

Syntax
stop

Description
Stops program execution until a reset occurs. This is an alias to END.

Example
The following example will only run once. The program will only restart if the unit is reset.

```
value var long

      puts 0,[“This program just stops.”,13]
      puts 0,[“Press reset to see it again.”,13]
stop
```

WHILE - WEND

Syntax
while condition
program statements
wend

- **Condition** - can be a variable or expression

- **Statements -** any group of commands to be run inside the loop.

Description
The WHILE - WEND loop executes commands nested inside of it while some condition is true. The condition is tested before the WHILE - WEND loop is run. This is opposite of DO - WHILE which will test the condition for true after running once. The condition can be any variable or expression and is tested every loop until false. A simple example would be using the WHILE - WEND loop to test the state of an input pin. If the pin is low "go do something" until pin is high.

Notes
The loop will continue until its value equals 0.
WHILE - WEND checks the condition first. If the condition is false the WHILE - WEND statements and all program code nested within them will not run.
You can nest multiple WHILE - WEND commands within each other. However, you can not nest DO - WHILE with a WHILE - WEND together or the compiler will get the WHILE statements confused.

Example
Connect to the following program with the terminal window set to 9600 baud. The program will start counting up from 0 to 100. Once index reaches a value of 100 the condition is no longer true. The less than symbol (<) was used for the condition and 100 is no longer less than 100 making the condition false. Since WHILE - WEND loops if a statement is true the program exits.

```
;ALL - all_while_wend2.bas

Index var word

Main
       Index = 0

       While Index < 100 ;run until no longer less than 100
       index = index + 1

       ;print the value of index
       puts 0,[0, "Couting: ", dec index]
       pause 75

       wend

       puts 0, [13,13, "Index = ", dec index]
       puts 0, [13, "My condition is no longer true."]
       puts 0, [13, "Index is no longer less than 100"]
End
```

WRITE

Syntax
write address, data

- **Address** - address of EEPROM to store data

- **Data** - is a byte value that will be written to the address specified. It can be an expression, constant or variable.

Description
All modules except the BasicATOM Pro 24 come with built in EEPROM. The READ / WRITE commands were created to access the built-in memory. WRITE will write a signal byte to the built-in EEPROM at the address specified in the command.

Example
The example program will write the string "Hello" starting at the first location to the built-in memory. Next, it will read built-in memory locations 0 through 10 and print the contents to the terminal window.

```
index var byte
char var byte

write 0,"H"
write 1,"e"
write 2,"l"
write 3,"l"
write 4,"o"

for index = 0 to 10
      read index,char
      puts 0,[char]
next
end
```

3.4 Compile Time Directives

3.4.1 Compiler Directives

MCL supports compile time directives. Compile time directives can be used to selectively include or exclude parts of a program which can be very useful. During compile time you can define parts of the program to compile or not.

3.4.2 Conditional Compiling

If you've written a program to display temperature from a sensor, you may want versions for Celsius and Fahrenheit. Most of the code is identical, but some constants, variables and subroutines may differ.

Conditional compiling lets you set a "switch" in your program that controls compiling. You can have different constants, variables, or even different sections of code compiled depending on the switch or switches that you set.

#IF .. #ENDIF

#IF expression
optional code
#ENDIF

Example

Similar to IF..THEN conditional branch, but specifies code to be compiled if the expression is true. In the example below the constant temp is set to 1. During compile time the #IF will test temp to see if it is true. In the example below will return true so the following block of code is included during compile time.

```
temp con 1
#IF temp=1
      ..optional code..
#ENDIF
..rest of program..
```

#IFDEF .. #ENDIF
#IFDEF name
..optional code..
#ENDIF
..rest of program..

Example
Compiles the code (up to #ENDIF) if the constant or variable (name) is defined, or if the label appears previously in the code.

```
temperature var byte
#IFDEF temperature
       ..optional code..
#ENDIF
..rest of program..
```

This will compile “optional code” because “temperature” has been defined.

#IFNDEF .. #ENDIF
#IFNDEF name
..optional code..
#ENDIF
..rest of program..

Example
Compiles the code between #IFNDEF and #ENDIF only if the constant or variable has not been defined, or the label has not been previously used in the program. In effect, it is the opposite of #IFDEF.

```
temperature var byte
#IFDEF temperature
       ..optional code..
#ENDIF
..rest of program..
```

This will NOT compile “optional code” because “temperature” has been defined.

#ELSE
#IF expression
..optional code..
#ELSE
..use this code if the other code wasnt used..
#ENDIF
..rest of program..

Example
Allows you to have two code snippets and compile one or the other depending on the result of the #IF, #IFDEF or #IFNDEF directive.

```
temp con 1
#IF temp=1
       ..optional code..
#ELSE
       ..more optional code..
#ENDIF
       ..rest of program..
```

Compiles “optional code” if “temp = 1” and “more optional code” if “temp” is equal to any other value.

#ELSIF
#IF expression
..optional code..
#ELSEIF
..more optional code..
#ELSEIF
..even more optional code..
#ELSE
..if nothing else was used, then use this code..
#ENDIF
..rest of program..

Example
Allows multiple snippets of code to be compiled based on multiple tests. ELSEIF is an extension of #ELSE and allows multiple conditional tests to run during compile time.

```
screentype con 1

#IF screentype=1
       ..optional code..
#ELSEIF screentype=2
       .. more optional code..
#ELSEIF screentype=3
       ..even more optional code..
#ENDIF
... rest of program ...
```

Compiles "optional code", "more other code", or "even more optional code" depending on what the constant "screentype" is set to (1, 2 or 3). If "screentype" has some value other than 1,2 or 3 compilation simply continues with "rest of program" and none of the optional code is compiled.

#ELSEIFDEF, #ELSEIFNDEF
#ELSEIFDEF name
#ELSEIFNDEF name

Example
Equivalents of #ELSEIF for the #IFDEF and #IFNDEF directives. Similar to the example given for #ELSEIF.

ASCII Table

Dec	Hex	Char	Function
0	0x00	NUL	Null
1	0x01	SOH	Start of Heading
2	0x02	STX	Start of Text
3	0x03	ETX	End of Text
4	0x04	EOT	End of Transmit
5	0x05	ENQ	Enquiry
6	0x06	ACK	Acknowledge
7	0x07	BEL	Bell
8	0x08	BS	Basckspace
9	0x09	HT	Horizontal Tab
10	0x0A	LF	Line Feed
11	0x0B	VT	Vertical Tab
12	0x0C	FF	Form Feed
13	0x0D	CR	Carriage Return
14	0x0E	SO	Shift Out
15	0x0F	SI	Shift In
16	0x10	DLE	Data Line Escape
17	0x11	DC1	Device Cntrl 1
18	0x12	DC2	Device Cntrl 2
19	0x13	DC3	Device Cntrl 3
20	0x14	DC4	Device Cntrl 4
21	0x15	NAK	Non Acknowledge
22	0x16	SYN	Synchronous Idle
23	0x17	ETB	End Transmit Block
24	0x18	CAN	Cancel
25	0x19	EM	End of Medium
26	0x1A	SUB	Substitute
27	0x1B	ESC	Escape
28	0x1C	FS	File Separator
29	0x1D	GS	Group Separator
30	0x1E	RS	Record Separator
31	0x1F	US	Unit Separator
32	0x20	SPACE	
33	0x21	!	
34	0x22	"	
35	0x23	#	
36	0x24	$	
37	0x25	%	
38	0x26	&	
39	0x27	'	
40	0x28	(	
41	0x29	)	
42	0x2A	*	

Dec	Hex	Char
43	0x2B	+
44	0x2C	,
45	0x2D	-
46	0x2E	.
47	0x2F	/
48	0x30	0
49	0x31	1
50	0x32	2
51	0x33	3
52	0x34	4
53	0x35	5
54	0x36	6
55	0x37	7
56	0x38	8
57	0x39	9
58	0x3A	:
59	0x3B	;
60	0x3C	<
61	0x3D	=
62	0x3E	>
63	0x3F	?
64	0x40	@
65	0x41	A
66	0x42	B
67	0x43	C
68	0x44	D
69	0x45	E
70	0x46	F
71	0x47	G
72	0x38	H
73	0x49	I
74	0x4A	J
75	0x4B	K
76	0x4C	L
77	0x4D	M
78	0x4E	N
79	0x4F	O
80	0x50	P
81	0x51	Q
82	0x52	R
83	0x53	S
84	0x54	T
85	0x55	U

Dec	Hex	Char
86	0x56	V
87	0x57	W
88	0x58	X
89	0x59	Y
90	0x5A	Z
91	0x5B	[
92	0x5C	\
93	0x5D	]
94	0x5E	^
95	0x5F	_
96	0x60	`
97	0x61	a
98	0x62	b
99	0x63	c
100	0x64	d
101	0x65	e
102	0x66	f
103	0x67	g
104	0x68	h
105	0x69	i
106	0x6A	j
107	0x6B	k
108	0x6C	l
109	0x6D	m
110	0x6E	n
111	0x6F	o
112	0x70	p
113	0x71	q
114	0x72	r
115	0x73	s
116	0x74	t
117	0x75	u
118	0x76	v
119	0x77	w
120	0x78	x
121	0x79	y
122	0x7A	z
123	0x7B	{
124	0x7C	|
125	0x7D	}
126	0x7E	~
127	0x7F	Delete

Warranty

Basicmicro warranties its products against defects in material and workmanship for a period of 1 year. If a defect is discovered, Basicmicro will, at our sole discretion, repair, replace, or refund the purchase price of the product in question. Contact us at sales@basicmicro.com. No returns will be accepted without the proper authorization.

Copyrights and Trademarks

Disclaimer

Basicmicro cannot be held responsible for any incidental or consequential damages resulting from use of products manufactured or sold by Basicmicro or its distributors. No products from Basicmicro should be used in any medical devices and/or medical situations. No product should be used in any life support situations.

Contacts

Email: sales@basicmicro.com
Tech support: support@basicmicro.com
Web: http://www.basicmicro.com

Discussion List

A web based discussion board is maintained at http://www.basicmicro.com

Technical Support

Technical support is available by sending an email to support@basicmicro.com, by opening a support ticket on the Ion Motion Control website or by calling 800-535-9161 during normal operating hours. All email will be answered within 48 hours.