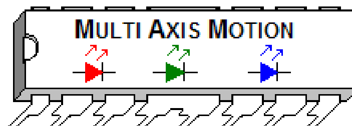
Prepared By

Arnold Stadlin

MULTIAXISMOTION.COM



121 W. Earleigh Heights Rd.
Severna Park, MD 21146
410-315-7590

Support and Guidance Provided By

The EET/EGR Faculty of Anne Arundel Community College

Arnold, Maryland

For additional project information and related topics; please visit
www.ModernControlTechnology.com

Revision	Description
12/23/07.0014	MOTION HOME SpeedMinimum parameter added, GOTO program command added
12/21/07.0013	MOTOR PWMPINS Instruction
12/20/07.0012	MOTION HOME SpeedLimit parameter added
12/16/07.0009	MOTOR GPWMn, PWMn, and MOTION START enhanced
12/01/07.0007,8	MOTION INIT, HOME, DIR, GOTO Instructions; Program RESTART
11/30/07.0006	MOTION START and STOP Instructions
11/28/07.0005	MOTOR PWM Instructions
11/27/07.0004	PAUSE Instruction
11/27/07.0003	MOTOR PINS Instruction
11/25/07.0002	MOTOR GPORT Instructions
11/18/07	POSITION Instructions
11/17/07	PING Instruction
11/13/07	MCL Interpreter; Derived from the Pegasus Robot Motion Controller Documentation

Table of Contents

Document Summary....................................3
Motion Control System - MCS....................................4
Communications Protocol....................................4
Message Template....................................6
Gateway Packet Translation....................................6
Motion Controller Messages....................................7
Communications Process....................................9
Under the Hood....................................9
Motion Control Language....................................10
Motion Control Language Instruction Set....................................11
MOTION Instructions....................................11
MOTION MCU_ID INIT....................................12
MOTION MCU_ID HOME Action SpeedLimit SpeedMinimum....................................13
MOTION MCU_ID GOTO NavSector....................................15
MOTION MCU_ID START Direction Speed_Preset....................................16
MOTION MCU_ID STOP....................................16
MOTION MCU_ID DIRECTION Direction....................................17
MOTION MCU_ID DIRSPEED Direction Speed_Preset....................................17
MOTION MCU_ID SPEED Speed_Preset....................................17
MOTOR Instructions....................................18
MOTOR MCU_ID GPORTn....................................18
MOTOR MCU_ID GPWMn....................................19
MOTOR MCU_ID PWMn PTPER PDC PENHL BreakEnable....................................20
MOTOR MCU_ID PWMPINS PWM1 PWM2 BreakEnable....................................21
MOTOR MCU_ID PINS1 [Mask_byte] [Value_byte] [Operation]....................................22
PAUSE <MilliSeconds>....................................23
RESTART....................................24
GOTO....................................25
PING MCU_ID [Count] [Delay]....................................26
POSITION MCU_ID [NAV | QEI] [TimeOut]....................................27
MCL Interpreter....................................28
Motor Control Registers....................................29
Glossary....................................30

DOCUMENT SUMMARY

This document is a supplement to the Multi Axis Motion Controller Documentation for the Pegasus Robot Project. The scope of this document is limited to the implementation of a motion control instruction set and interpreter. In this document, the overall system is called the *Motion Control System*, or *MCS*.

The Motion Control Language, or *MCL*, developed here is designed to be interpreted by a PC host and transmitted via RS232 to a micro controller gateway on the motion controller's CAN - Controller Area Network. The micro controllers of the motion controller are CAN *Nodes.* Details of the UART (RS232) and CAN implementation are found in the Project documentation.

MOTION CONTROL SYSTEM - MCS

The *Motion Control System,* or *MCS,* for the Pegasus Robot consists of the following general components:

- Motors: Pedestal Robot Arm Axis motors, Conveyor, and Index Table.
- Micro Controller Motor Controllers: CAN Networked dsPIC30F4011 micro controllers
- Micro Controller CAN to RS232 Gateway
- Workstation PC: Human Machine Interface, Motion Control Language, and Interpreter and Software

COMMUNICATIONS PROTOCOL

The following table shows how the Controller Area Network, or *CAN,* and the Motion Control System map to the OSI network model.

OSI	Motion Control System and Controller Area Network	
Application	MCS MCL	
Presentation	MCS MCL Interpreter	
Session	MCS Communications Protocol	
Transport	MCS Communications Protocol	
Network	MCS Communications Protocol	dsPIC CAN Peripheral
Data Link Layer	Logical Link Control Medium Access Control (MAC)	dsPIC CAN Peripheral
Physical Layer	Physical Signalling, Bit Encoding	MCP2551 CAN Transceiver

Before deciding on a protocol for the motion controller network, I reviewed CANOpen and DeviceNet. Although both are industry standards, neither are publicly documented well enough to implement easily. The specifications are commercially available for a cost beyond the scope of this project. For this motion controller, we will develop our own protocol with some features similar to DeviceNet but much simpler overall.

The CAN protocol itself consists of two parts from which we will develop our communications protocol: a Standard Identifier (11 bits) and 8 bytes of data. We will split the SID into 2 segments. The MSB 4 bits will be used as the Target Micro Controller's unique ID, or *MCU_ID*. The remaining 7 bits will be used for Message types.

The prefixes *RX* = Target Receiving and *TX* = Source Transmitting are used in the context of the Micro Controller sending the message. The *MCU_ID* = 0 for the top nibble is for broadcasting a message to ALL micro controllers. *HI* and *LO* are most and least significant bytes respectively. *hi* and *lo* are most and least significant nibbles (4 bits each) respectively.

CAN Standard Identifier = 11 bits	
SID<10..7> (4 bits)	**SID<6..0> (7 bits)**
Target Mico Controller's MCU ID	Message Classes

Our Data packet will provide the Transmitting Micro Controller's ID (4 bits), Message Category (4 bits), Function number (1 byte), and data (6 bytes). The Data in the packet is determined by the Message Category and Function's requirements.

4 Words (8 Bytes)	**HI Byte**	**LO Byte**
1	TX MCU_ID and Category	Instruction
2	data	data
3	data	data
4	data	data

In our high level code, we will consider the CAN SID to be a 16 bit unsigned integer masking and shifting the bits as needed. The micro controllers read and write the data as 16 bit words (unsigned integers). The RS232 received packet consists of a prefix <stx> start of packet byte, an unsigned 32bit integer for a Packet Sequence ID followed by 5 unsigned 16bit integers for the CAN SID (1 word), CAN Data (4 words), a CRC (1 word), and a <etx> end of packet byte. If we find data errors to be intolerable, we will add a CRC word to the packets. The following is the message template.

```
<stx>PacketID;CANSID;data;data;data;data;CRC<etx>
 1 +     2 +      2 +  2 +  2 +  2 +  2 + 2 + 1   = 16 bytes packet size
```

Message Template

The following is a message template. For RS232 messages received by the PC, there is a PacketID prefix and a CRC suffix. For all RS232 messages there is a <stx> and <etx> at the beginning and end of packet. Since these are always present, they are not included in the template.

	HI Byte		LO Byte		
Word	**[15..12]**	**[11..8]**	**[7..4]**	**[3..0]**	**Description**
SID	SID_MCUID	[0x0]	MSG_ID		Target MCU ID (4 bits) and Message Class ID (7 LSB bits)
0	TX_MCUID	TXCAT	INSTRUCTION		Trasmitting MCU ID, Category, and Instruction
1	Data Byte[0]		Data Byte[1]		2 Data bytes
2	Data Byte[2]		Data Byte[3]		2 Data bytes
3	Data Byte[4]		Data Byte[5]		2 Data bytes

Gateway Packet Translation

The CAN and RS232 packets are exchanged between the PC and the micro controllers, the Gateway MCU adds the <stx>, PacketID, CRC, and <etx> to the packets sent to the PC and strips it from the packets received from the PC.

```
C1RX0SID Receive CAN Standard Identifier
UNUSED uuu. .... .... ....
   SID ...0 0000 0000 00..  Standard Indentifier, 11 bits
   SRR .... .... .... ..0.  Substitute Remote Request; 1=Remote Trsnafer Requested
 RXIDE .... .... .... ...0  Extended Identifier; 1=Extended ID, 0=Standard ID

C1TX0SID Transmit CAN Standard Identifier
UNUSED .... .uuu .... ....
   SID 0000 0... 0000 00..  Standard Identifier, 11 Bits
   SRR .... .... .... ..0.  Substitute Remote Request Control
 TXIDE .... .... .... ...0  Transmit Extended Identifier
```

The PC extracts the protocol info from a Recevied CAN SID

```
// C1RXnSID[HI] = aBytes[3]; C1RXnSID[LO] = aBytes[4]
SID_McuId = (byte)((aBytes[3] & 0x1E) >> 1);      // Mask HI 4 SID bits; shift 1 right
SID_MsgClass = (byte)((aBytes[4] & 0xFC) >> 2);   // Mask LO 6 SID bits; shift 2 right
SID_MsgClass += (byte)((aBytes[3] & 0x01) << 7);  // Mask HI 5th SID bit; sift left 7, Add to LO SID
TX_McuId = (byte)(aBytes[5] & 0xF0);
TX_Cat = (byte)(aBytes[5] & 0x0F);
TX_Inst = aBytes[6];
```

The PC encodes the protocol info to a Transmitted CAN SID

```
C1TX0SID = (RX_MCU_ID << 11) + (MSG_ID << 2)
```

MOTION CONTROLLER MESSAGES

The SID is 11 bits. For our messages, the upper 4 bits of the SID are the *MCU_ID* (micro controller identifier) and the lower 7 bits are the *Message Class.* We'll use the DeviceNet convention and view the Request VS Response from the perspective of the *Server* (the micro controller that transmits the packet). A Request message is *Downloaded* to *Clients* and a Response message is *Uploaded* to the *Server(s).* The Transmit Category, *TxCat,* is set to 0x01 for Instructions (including broadcasts), 0x02 for Replies, and 0x03 for Errors.

The *MCU_ID* = 0x0 is used to broadcast the message to all the micro controllers. The Gateway micro controller has 2 interfaces. The Gateway's RS232 interface, *UART_MCUID,* is 0xF0. The Gateway micro controller's CAN interface, *CAN_MCUID,* is 0x10. 0xF0 is also the PC's *MCU_ID.* The remaining *MCU_ID* values, 0x20 to 0xD0, are available for the individual micro controllers to use. The 0xE0 *MCU_ID* is used in combination with *Message Classes* to identify additional CAN Nodes; e.g. sensor nodes.

We will use the following conventions: Messages are prefixed with “MSG_”. Instructions are prefixed with “IX_[aaaa]”; “aaaa” is 4 letters related to the Message. The prefixes *RX* = Target Receiving and *TX* = Source Transmitting are used in the context of the Micro Controller sending the message (the *Server*). *HI* and *LO* are most and least significant bytes respectively. *hi* and *lo* are most and least significant nibbles (4 bits each) respectively.

Category TxCat	Description
CAT_EXEC	Execute the Instruction on the Target Node
CAT_REPLY	Respond to a Request Instruction
CAT_ERROR	Error Messages

Message Class	Instruction	Description
MSG_CLEAR		No Message, Cleared

MSG_STATUS		Status Messages
	IX_STAT_PING	Request On-Line Status

MSG_MOTION		Motion Control Messages
	IX_MHOME	Go to Home Position
	IX_MSTART	Execute the START Motor procedure with Direction and Speed Preset
	IX_MSTOP	Execute the STOP Motor procedure
	IX_MDIR	Set or Change the Motor's Motion Direction
	IX_MSPEED	Set or Change the Motor's Speed by Preset
	IX_MDIRSPEED	Set or Change the Motor's Direction and Speed

MSG_MOTOR		Motor Messages
	IX_MOTR_GPORT1	Get the MCU Port Pin Registers C and D
	IX_MOTR_GPORT2	Get the MCU Port Pin Registers E and F
	IX_MOTR_GPWM1	Get the MCU PWM Registers *PTPER* and *PDC1*
	IX_MOTR_GPWM2	Get the MCU PWM Registers *PTPER* and *PDC2*
	IX_MOTR_PWM1	Set the MCU PWM Registers *PTPER* and *PDC1*
	IX_MOTR_PWM2	Set the MCU PWM Registers *PTPER* and *PDC2*
	IX_MOTR_PWMPINS	Set the MCU PWM *PDC1, PDC2*, Break, and *PTEN*
	IX_MOTR_PINS1	Set the Motor Controller's Enable, Phase, Break, and Mode pins
	IX_MOTR_RUN	Execute the Motor Controller's RUN procedure with PWM & PINS
	IX_MOTR_STOP	Execute the Motor Controller's STOP procedure with PWM & PINS

MSG_POSITION		Positioning Messages
	IX_POSI_NAV	Get Navigable Space Logical Coordinates
	IX_POSI_QEI	Get Quadrature Encoder Physical Values

MSG_SENSOR		Sensor Messages

MSG_ERROR		Error Messages
	IX_CAN_ERR	CAN Error Message

COMMUNICATIONS PROCESS

The table below shows example communications between the PC and the CAN

PC RS232; ID = 0xF0	Gateway RS232	Gateway CAN; ID = 0x10	CAN Node; ID = 0x20
SID_MCUID = 0x10 TX_MCUID = 0xF0	Send Message to Gateway	Execute Instruction	Ignore
SID_MCUID = 0x20 TX_MCUID = 0xF0	Relay PC to CAN 0x20	Ignore	Execute Instruction
Execute Instruction	Send Message to PC	SID_MCUID = 0xF0 TX_MCUID = 0x10	Ignore
Execute Instruction	Relay CAN to PC 0xF0	Ignore	SID_MCUID = 0xF0 TX_MCUID = 0x20
Diagnostic Listen Only	Diagnostic Mode Only Relay	SID_MCUID = 0x20 TX_MCUID = 0x10	Execute Instruction
Diagnostic Listen Only	Diagnostic Mode Only Relay	Execute Instruction	SID_MCUID = 0x10 TX_MCUID = 0x20
SID_MCUID = 0x00 TX_MCUID = 0xF0	Broadcast PC Message to CAN 0x00	Execute Instruction	Execute Instruction
Execute Instruction	Relay CAN to PC 0xF0	SID_MCUID = 0x00 TX_MCUID = 0x10	Execute Instruction
Execute Instruction	Relay CAN to PC 0xF0	Execute Instruction	SID_MCUID = 0x00 TX_MCUID = 0x20

Under the Hood

Each micro controller, or *MCU,* has 2 network interfaces: UART (RS232) and CAN. However, only one of the MCUs uses the UART to act as a Gateway between the PC and the CAN. The protocol for this project uses packets that can easily be relayed between the UART and CAN interfaces on the Gateway MCU.

The *Process_UART* procedure, running on the Gateway MCU, decodes packets sent from the PC to the CAN. If the Packet's SID_MCUID is the Gateway's MCU_Id, the packet is handled by the Gateway; if not, then the packet is relayed to the CAN.

The *Process_CAN* procedure, running on each MCU, decodes packets it receives from the CAN. If the packet's SID_MCUID is the MCU's ID, then the MCU processes the packet. If the SID_MCUID is for the PC and the MCU is the Gateway MCU, then the MCU relays the CAN packet to the UART.

MOTION CONTROL LANGUAGE

Using a scripting language permits multiple command tasks to be submitted to the Motion Controller automatically with or without operator interaction. Since our Motion Controller and prototype are limited in scope, we will develop our own programming language and runtime interpreter. Implementing any significant compatibility with existing motion control languages is beyond the scope of this project.

The MCS (Motion Control System) executes MCL - Motion Control Language programs. The MCL has the following simple syntax consisting of a Command, Target MCU_ID, and Command Parameters. The Command line is delimited by spaces. A ";" semicolon can be placed in the command or at the start of a line to designate the remainder of the line's text as a comment.

COMMAND MCU_ID [Param0] [Param1] [...] ; Comments after ";" character

The MCL Interpreter, *MCLI*, will ignore leading and trailing spaces.

Although the interpreter may sometimes ignore character case, assume it is case sensitive. By convention, all commands and constants are UPPER CASE. Variables are mixed case.

Parameters denoted in "[ ]" brackets are optional.

All numeric parameters should be entered in "C" style Hexadecimal. e.g. 0xF0

Motion Control Language Instruction Set

MOTION Instructions

MOTION instructions are high level instructions for the motor controllers used for automated control. The *MOTION* instructions are transmitted to the motor controllers which then execute their built in procedures for handling the instructions. For more direct control of the motor controller MCU's, use the *MOTOR* instructions.

Typically the *MOTION* instructions will override the *MOTOR* instructions and vice versa; which ever are executed last. For example you can send the *MOTION 0x20 START 1 0* instruciton to start motor 0x20 at slow speed, then gradually increase the speed using *MOTOR 0x20 PWM* commands.

It is a better practice to use the *MOTION* instructions instead of the *MOTOR* instructions for most tasks as they are more generic than the *MOTOR* instructions. The *MOTION* instructions are designed to work for all the motor controllers. For example, *MOTION DIR* sets the direction of motion; however motion direction may not be controllable by setting the *MOTOR PINS* phase pin in every motor.

MOTION MCU_ID *INIT*

INIT resets the motor controller to its startup configuration. The *INIT* instruction does not execute the *Homing* procedure of the controller.

MOTION *MCU_ID INIT*			
SID	SID_MCU_ID	0x0	***MSG_MOTION***
0	TX_MCU_ID	CAT_EXEC	***IX_MINIT***
1			
2			
3			

Example

```
; Synchronously Reset Motor Controller #2
MOTION 0x20 INIT
; Asynchronously Reset all of the Motor Controllers
MOTION 0x00 INIT
```

The *MOTION INIT* instruction is Asynchronous; there is no reply packet.

MOTION MCU_ID *HOME Action SpeedLimit SpeedMinimum*

MOTION MCU_ID AHOME *Action SpeedLimit SpeedMinimum*

Revised 12/23/07.0014

HOME instructs the motor controller to execute its *Homing* procedure. Typically, the *Homing* procedure sends the motor to a position that triggers an optical interrupter or limit switch. *HOME* is the absolute origin in the motor's coordinate system and navigational space. When the motor hits the *HOME* position, the Quadrature Encoder registers and Navigational coordinates are reset to zero, and if applicable, translated with additional offsets specific to each motor.

The *Action* parameter is used to specify additional options. *Action = 1* is the standard system startup *Homing* procedure. The *Actions* are described in the table below. *Actions* that *Stop* are synchronous and reset the *Odometer*. The “don't Stop” *Actions* are asynchronous and used when you want to take direct control of the motor using *MOTOR* commands before the *Homing* routine completes. *Actions* that don't stop increment or decrement the *Odometer* depending on direction.

The *SpeedLimit* parameter is the maximum PWM duty cycle %. For example *Speed* = 0x64 is 100% duty cycle.

The *SpeedMinimum* parameter is the minimum PWM duty cycle %. *SpeedMinimum* is used to maintain the motion in an “energized” state. When using *PHASE Modulation*, this energized state helps to lock the motor into position while using a minimum *ENABLE* cycle reduces oscillation around the position.

Action	Description
0	Go Home and Stop with current motor running parameters
1	Initialize Motor, Go Home, Stop
2	Go Home with current motor running parameters, don't Stop
3	Initialize Motor, Go Home, don't Stop

MOTION *MCU_ID HOME Action SpeedLimit SpeedMinimum*			
SID	SID_MCU_ID	0x0	MSG_MOTION
0	TX_MCU_ID	CAT_EXEC	***IX_HOME***
1	***Action***		***SpeedLimit***
2	***SpeedMinimum***		
3			

Example

```
; Synchronously execute Controller 0x20's Homing procedure
MOTION 0x20 HOME 1 0x64 0x20
; Execute the Homing routines on all controllers
MOTION 0x00 HOME 1 0x64 0x20
```

For MCU_ID not equal 0x00, the *MOTION HOME* instruction is synchronous. The *MOTION AHOME* instruction is asynchronous. Broadcasts to 0x00 are asynchronous after the first MCU responds with a *IX_POSI_QEI* packet. The *MOTION HOME* and *AHOME* instructions returns two position packets. The values returned are saved in the same registers as the *POSITION* registers in the following UInt16 arrays:

```
MotPosi_Nav[16]    ; Navigable Space Position
MotPosi_Qei[16]    ; Quadrature Encoder Position
MotPosi_Sec[16]    ; Position Sector
MotPosi_Dir[16]    ; Motor Direction: 0 = Forward, 1 = Reverse or vice-versa
MotPosi_Odo[16]    ; Odometer (number of times crossing the Home position.
```

SID	SID_MCU_ID	0x0	MSG_MOTION

0	**TX_MCU_ID**	**CAT_RPLY**	**IX_POSI_NAV**
1	***Nav_Position***[HI]		***Nav_Position***[LO]
2	***Direction***		***Sector***
3	***Odometer***[HI]		***Odometer***[LO]

SID	SID_MCU_ID	0x0	MSG_MOTION
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_POSI_QEI**
1	***QEI_Position***[HI]		***QEI_Position***[LO]
2	***Direction***		***Sector***
3	***Odometer***[HI]		***Odometer***[LO]

Note: The *SpeedLimit* is used by the *MOTION* commands, not the *MOTOR* commands. To override the *SpeedLimit*, use the *MOTOR PWM* commands for speed and phase control.

MOTION *MCU_ID GOTO NavSector*

GOTO instructs the motor controller to execute its *Motion_GoTo* procedure. *NavSector* is the logical coordinate to traverse to.

MOTION *MCU_ID GOTO NavSector*			
SID	SID_MCU_ID	0x0	MSG_MOTION
0	TX_MCU_ID	CAT_EXEC	***IX_GOTO***
1	***NavSector***		
2			
3			

Example

```
; This program rotates an Index Table 45 deg ever 2 seconds
MOTION 0x20 HOME 1
PAUSE 2000
MOTION 0x20 GOTO 0x00B4
PAUSE 2000
MOTION 0x20 GOTO 0x0168
PAUSE 2000
MOTION 0x20 GOTO 0x021C
PAUSE 2000
MOTION 0x20 GOTO 0x02D0
PAUSE 2000
MOTION 0x20 GOTO 0x0384
PAUSE 2000
MOTION 0x20 GOTO 0x0438
PAUSE 2000
MOTION 0x20 GOTO 0x04EC
PAUSE 2000
RESTART
```

The *MOTION GOTO* instruction is asynchronous. When the controller reaches the target *NavSector* it will transmit two position packets. The values returned are saved in the same registers as the *POSITION* registers in the following UInt16 arrays:

```
MotPosi_Nav[16]    ; Navigable Space Position
MotPosi_Qei[16]    ; Quadrature Encoder Position
MotPosi_Sec[16]    ; Position Sector
MotPosi_Dir[16]    ; Motor Direction: 0 = Forward, 1 = Reverse or vice-versa
MotPosi_Odo[16]    ; Odometer (number of times crossing the Home position.
```

SID	SID_MCU_ID	0x0	MSG_MOTION
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_POSI_NAV**
1	***Nav_Position***[HI]		***Nav_Position***[LO]
2	***Direction***		***Sector***
3	***Odometer***[HI]		***Odometer***[LO]

SID	SID_MCU_ID	0x0	MSG_MOTION
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_POSI_QEI**
1	***QEI_Position***[HI]		***QEI_Position***[LO]
2	***Direction***		***Sector***
3	***Odometer***[HI]		***Odometer***[LO]

MOTION *MCU_ID START Direction Speed_Preset*

MOTION *MCU_ID STOP*

START Instructs the motor controller to execute its Start Motor procedure in *Direction* of ***0*** or ***1*** at *Speed_Preset*. *Direction* is 0/1 = Forward/Reverse and is arbitrary with respect to each motor controller's configuration. Speed presets are positive integers and may vary for the different controllers. Speed = **0** is minimum. The Index Table and Conveyor may have a maximum speed of **4**. The following is a sample table of *Speed_Preset* calculations.

```
0: PDC1 = (unsigned int)(PTPER * 1.20); // Minimum
1: PDC1 = (unsigned int)(PTPER * 1.40); // Slow
2: PDC1 = (unsigned int)(PTPER * 1.60); // Medium
3: PDC1 = (unsigned int)(PTPER * 1.80); // Fast
4: PDC1 = (unsigned int)(PTPER * 2.00); // Maximum
```

MOTION *MCU_ID START Direction Speed_Preset*			
SID	SID_MCU_ID	0x0	***MSG_MOTION***
0	TX_MCU_ID	CAT_EXEC	***IX_MSTART***
1	***Direction***		***Speed_Preset***
2			
3			

MOTION *MCU_ID STOP*			
SID	SID_MCU_ID	0x0	MSG_MOTION
0	TX_MCU_ID	CAT_EXEC	***IX_MSTOP***
1			
2			
3			

<u>Example</u>

```
; Stop ALL the Motors on ALL controllers
MOTION 0x00 STOP
; Run the Motor 2 Forward, Speed 4, for 1 second
MOTION 0x20 START  0  4
PAUSE 1000
; Run the Motor 2 Reverse, Speed 4, for 1 second
MOTION 0x20 START  1  4
PAUSE 1000
; Stop ALL the Motors on ALL controllers
MOTION 0x00 STOP
```

The *START* and *STOP MOTION* instructions are Asynchronous. There is no reply packet.

When running in the *Forward Direction*, the *Odometer* increments when the motor passes the *Home* position. In the *Reverse Direction*, the *Odometer* decrements when the motor passes the *Home* position.

Observation Note: For the motors we are using to test this application and a 12V DC power supply; a duty cycle of 50% is not adequate to drive the motor. The PTPER register value + 1 = 50% duty cycle. The minimum speed to overcome the gear motor friction is about 1.2x the PTPER register value. 2x the PTPER register value = 100% duty cycle.

MOTION MCU_ID *DIRECTION Direction*

MOTION MCU_ID *DIRSPEED Direction Speed_Preset*

MOTION MCU_ID *SPEED Speed_Preset*

DIRECTION sets the motion direction, *SPEED* sets the motion speed. *DIRSPEED* sets the *Direction* and *the Speed*. *Direction =* ***0*** is *Forward*, and ***1*** is *Reverse*. Forward/Reverse is arbitrary with respect to each motor controller's configuration. *Speed_Preset* is a number representing a pre-programmed speed. *Speed_Preset = 0* is minimum speed (not Stopped).

MOTION ***MCU_ID DIRECTION Direction***			
SID	SID_MCU_ID	0x0	***MSG_MOTION***
0	TX_MCU_ID	CAT_EXEC	***IX_MDIR***
1	***Direction***		
2			
3			

MOTION ***MCU_ID DIRSPEED***			
SID	SID_MCU_ID	0x0	MSG_MOTION
0	TX_MCU_ID	CAT_EXEC	***IX_MDIRSPEED***
1	***Direction***		***Speed_Preset***
2			
3			

MOTION ***MCU_ID SPEED***			
SID	SID_MCU_ID	0x0	MSG_MOTION
0	TX_MCU_ID	CAT_EXEC	***IX_MSPEED***
1			***Speed_Preset***
2			
3			

Example

```
; Set Motor #2 to fast forward
MOTION 0x20 DIRSPEED 0 4
```

The *MOTION DIRECTION*, *SPEED*, and *DIRSPEED* instructions are Asynchronous. There is no reply packet.

When running in the *Forward Direction*, the *Odometer* increments when the motor passes the *Home* position. In the *Reverse Direction*, the *Odometer* decrements when the motor passes the *Home* position.

MOTOR INSTRUCTIONS

MOTOR instructions are used for direct interaction with the MCU motor controllers. These instructions read and write the I/O ports and MCU registers. They are particularly valuable for experimentation, diagnostics, and troubleshooting. For general motion control, the high level *MOTION* instructions should be used.

MOTOR MCU_ID GPORT*n*

GPORT1 gets the MCU's PORT B, C, and D Registers. *GPORT2* gets the MCU's PORT E and F Registers. The MCU Ports are returned as UInt16 values.

MOTOR *MCU_ID GPORTn*			
SID	SID_MCU_ID	0x0	MSG_MOTOR
0	TX_MCU_ID	CAT_EXEC	IX_MOTR_GPORT1 or *2
1			
2			
3			

<u>Example</u>

```
; Synchronously Query Motor Controller 0x10 for its I/O Port Registers
MOTOR 0x10 GPORT1
MOTOR 0x10 GPORT2
; Synchronously Query Motor Controller 0x20 for its I/O Port Registers
MOTOR 0x20 GPORT1
MOTOR 0x20 GPORT2
; Asynchronously Get All the Motor Control Registers
MOTOR 0x00 GPORT1
MOTOR 0x00 GPORT2
```

For MCU_ID not equal 0x00, the *MOTOR GPORT* instructions are synchronous. Broadcasts to 0x00 are asynchronous after the first MCU responds. The values returned are saved in the following UInt16 arrays:

```
MotPort_B[16]   ; MCU Register for PORT B
MotPort_C[16]   ; MCU Register for PORT C
MotPort_D[16]   ; MCU Register for PORT D
MotPort_E[16]   ; MCU Register for PORT E
MotPort_F[16]   ; MCU Register for PORT F
```

SID	SID_MCU_ID	0x0	MSG_MOTOR
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_MOTR_GPORT1**
1	***PORT B***		
2	***PORT C***		
3	***PORT D***		

SID	SID_MCU_ID	0x0	MSG_MOTOR
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_MOTR_GPORT2**
1	***PORT E***		
2	***PORT F***		
3			

MOTOR MCU_ID *GPWMn*

GPWM1 gets the MCU's PWM (Pulse Width Modulation) Registers. If the MCU has more than 1 PWM peripheral configured, then *GPWM2* will retrieve the next register set. The MCU Register values are returned as UInt16 values.

MOTOR *MCU_ID GPWMn*			
SID	SID_MCU_ID	0x0	MSG_MOTOR
0	TX_MCU_ID	CAT_EXEC	**IX_MOTR_GPWM1 or 2**
1			
2			
3			

Example

```
; Synchronously Query Each MCU's PWM Registers
MOTOR 0x10 GPWM1
MOTOR 0x20 GPWM1

; Asynchronously Query All of the MCUs' PWM Registers
MOTOR 0x00 GPWM1
```

For MCU_ID not equal 0x00, the *MOTOR GPWM* instructions are synchronous. Broadcasts to 0x00 are asynchronous after the first MCU responds. The values returned are saved in the following UInt16 arrays:

```
MotPWM_PTPER[16]   ; MCU Register for PWM1 Period
MotPWM_PDC1[16]    ; MCU Register for PWM1 Duty Cycle
MotPWM_PWMCON1[16] ; MCU Register for PWM Configuration
```

Received Packet

SID	SID_MCU_ID	0x0	MSG_MOTOR
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_MOTR_GPWMn**
1	***PWM_Period or PTPR***		
2	***PWM_DutyCycle or PDC1***		
3	***PWMCON1***		

Note: The PWM Period is typically constant for all PWM output for an individual MCU. PWM1_Period = PWM2_Period. The PWM Period defines the pulse frequency. The PWM Duty Cycle specifies the proportion of the period that is High or Low. PDC = (PTPER + 1) is about 50% Duty Cycle

MOTOR MCU_ID *PWMn PTPER PDC PENHL BreakEnable*

PWM1 sets the MCU's PWM (Pulse Width Modulation) Registers. If the MCU has more than 1 PWM peripheral configured, then *PWM2* will set the next register set. The MCU Register values are specified as UInt16 values in hexadecimal. PENHL sets the PWMCON.PENH and PWMCON.PENL. *BreakEnable* sets the BREAK pin and PTCON.PTEN bit.

MOTOR *MCU_ID PWMn*			
SID	SID_MCU_ID	0x0	MSG_MOTOR
0	TX_MCU_ID	CAT_EXEC	**IX_MOTR_PWMn**
1	***PTPER***		
2	***PDC***		
3	***PWMCON.PENH[1] + PENL[0]***	***BREAK[1] + PTCON.PTEN[0]***	

Example

```
; Set All PWM1 20kHz, 50% Duty Cycle, Enable Hi and Lo pins,and disable
MOTOR 0x00 PWM1 0x0577 0x0578  0x03 0x00

; Set the PWM1.PTPER, PDC, Release Break, and Enable MCU 0x20
MOTOR 0x20 PWM1 0x0577 0x0578  0x03 0x03
```

All *MOTOR PWMn* instructions are asynchronous.

Note: PWM1 and PWM2 share the same time base PTPER period.

MOTOR MCU_ID *PWMPINS PWM1 PWM2 BreakEnable*

12/21/07.0013

PWMPINS sets the MCU's PWM output pins. *PWM1* sets the *PWMCON.PEN1H* and *PWMCON.PEN1L* bits. *PWM2* sets the *PWMCON.PEN2H* and *PWMCON.PEN2L* bits. *BreakEnable* sets the BREAK pin and PTCON.PTEN bit.

MOTOR ***MCU_ID PWMn***			
SID	SID_MCU_ID	0x0	MSG_MOTOR
0	TX_MCU_ID	CAT_EXEC	***IX_MOTR_PWMPINS***
1	***PWM1***		***PWM2***
2	***BreakEnable***		
3			

Example

```
; Set All PWM Output Off, Disable PWM, set the Break (Active LO)
MOTOR 0x00 PWMPINS 0x00 0x00 0x00

; Set the PWM Output to PHASE Only for MCU 0x20, Enable PWM, release Break
MOTOR 0x20 PWMPINS 0x00  0x03 0x03
```

All *MOTOR PWMPINS* instructions are asynchronous.

MOTOR MCU_ID *PINS1* [*Mask_byte*] [*Value_byte*] [*Operation*]

PINS1 sets the MCU's Motor Controller Pins for direct program control ("manual mode") of the motor. On the Allegro A3953, there are 4 control pins: *Enable, Phase, Break, Mode*. There is also an *Activity* LED and an *Error* LED. Since there are only 6 bits required; the parameter is a single byte. The following table shows the bit layout of the *Pin_Set*. The *data_byte* is the bit values of the *Pin_Set.* At this time, bits 5 and 6 are not used.

	MSB 7	6	5	4	3	2	1	LSB 0
Output	*Error*	*x*	*x*	*Mode*	*Break*	*Phase*	*Enable*	*Active*
Mask	*0x80*	*0x40*	*0x20*	*0x10*	*0x08*	*0x04*	*0x02*	*0x01*

A bitwise *AND*, *OR*, *XOR*, or *SET Operations* can be performed by the output. *SET* is the default *Operation*. *SET* will set the Pins to the value specified in the *data_byte.*

MOTOR *MCU_ID PINS1 [data_byte] [Operation]*			
SID	SID_MCU_ID	0x0	MSG_MOTOR
0	TX_MCU_ID	CAT_EXEC	IX_MOTR_PINS1
1	***Pin_Set byte***		***data_byte***
2	***SET=0, AND=1, OR=2, XOR=3***		
3			

Example

```
; Set Clear all the Motor Control Pins for All the MCUs
MOTOR 0x00 PINS1 0xFF 0x00 SET
; Synchronize wait for MCU 0x20
PING 0x20
; Initialize Motor 0x20
;     Active_LED = HI,
;     Break_PIN = LO   (Break is Active Low)
;     Enable_PIN = HI  (Enable is Active Low)
MOTOR 0x20 PINS1 0x01 0x01 OR
; Release Break and !Enable; mask = 0x08 | 0x00 = 0x08
MOTOR 0x20 PINS1  0x08 0x08 OR
; The motor is now running!
```

The *MOTOR PINS1* instruction is output only; so by nature it is asynchronous and there will be no response from the MCU(s). Since there is no reply, a broadcast to MCU_ID 0x00 is received and executed by all the MCUs at the same time. However, it is a good idea to force the Interpreter to synchronize with MCUs for *MOTOR PINS1* instructions. This can be done with a *PING* instruction. The output values are saved in the following byte array:

```
MotPins_1[16]    ; MCU Motor Output Control Pins
```

PAUSE *<MilliSeconds>*

Pause the interpreter program execution for a period specified in milliseconds. The number of milliseconds is specified as a UInt32 (32 bit unsigned integer) in base 10. This pause is only pausing the interpreter's execution. The motor controllers do not pause; they continue their processing. The *PAUSE* command does not send any communications to the micro controllers.

Example

```
; Pause for 10 seconds
PAUSE 10000
```

The following screen shot shows the *MCLI* running a motor for 5 seconds in each direction. The *PAUSE* time remaining is displayed, counting down, in the bottom left corner of the *MCLI* window.

MCL Interpreter - ver 11/27/07.0004

File

On-Line | Registers

Received Packet

Rx Pkt #	4			Data	20	7
SID MCU_ID	f0	Msg	1		11	27
TX MCU	20 Cat 2	Inst	1		3	1

```
** Program Start **
MOTOR 0x00 PINS1 0xFF  0x00 SET
PING 0x20
-> RX# 4: 20[1], 20, 7, 11, 27, 3, 1
MOTOR 0x20 PINS1 0x01 0x03 OR
MOTOR 0x20 PINS1  0x08 0x08 OR
PAUSE 5000
```

C:\MuAxMo\RCGW\MCL\MOTOR_PAUSE_Test.mcl

```
; Set Clear all the Motor Control Pins for All the MCUs
MOTOR 0x00 PINS1 0xFF  0x00 SET

; Synchronize wait for MCU 0x20
PING 0x20

; Initialize Motor 0x20
;     Active_LED = HI,
;     Break_PIN = LO   (Break is Active Low)
;     Enable_PIN = HI  (Enable is Active Low)
MOTOR 0x20 PINS1 0x01 0x03 OR

; Release Break and !Enable; mask = 0x08 |0x00 = 0x08
MOTOR 0x20 PINS1  0x08 0x08 OR

; Run for 5 seconds
PAUSE 5000

; Reverse direction
MOTOR 0x20 PINS1  0x04 0x04 XOR
; Run for 5 seconds
PAUSE 5000

; Stop
; Set Break and Enable; mask = 0x08 |0x02 = 0x0A
MOTOR 0x20 PINS1  0x0A 0x0A OR

; Shut Down
MOTOR 0x20 PINS1 0xFF 0x00 SET
```

Program Paused... 00:00:03.097 | RX= 2 | TX= 11 | ERR= 0

RESTART

Restart the Program from the beginning.

Example

```
; This program rotates an Index Table 45 deg every 2 seconds
MOTION 0x20 HOME 1
PAUSE 2000
MOTION 0x20 GOTO 0x00B4
PAUSE 2000
MOTION 0x20 GOTO 0x0168
PAUSE 2000
MOTION 0x20 GOTO 0x021C
PAUSE 2000
MOTION 0x20 GOTO 0x02D0
PAUSE 2000
MOTION 0x20 GOTO 0x0384
PAUSE 2000
MOTION 0x20 GOTO 0x0438
PAUSE 2000
MOTION 0x20 GOTO 0x04EC
PAUSE 2000
RESTART
```

GOTO

Revised 12/23/07.0014

Arbitrary Goto a location in code and proceed execution from that point.

Example

```
; This program rotates an Index Table 45 deg ever 2 seconds
MOTION 0x20 HOME 1
PAUSE 2000
MOTION 0x20 GOTO 0x00B4
PAUSE 2000
MOTION 0x20 GOTO 0x0168
PAUSE 2000
MOTION 0x20 GOTO 0x021C
PAUSE 2000
; Label the 180 degree point
#180
MOTION 0x20 GOTO 0x02D0
PAUSE 2000
MOTION 0x20 GOTO 0x0384
PAUSE 2000
MOTION 0x20 GOTO 0x0438
PAUSE 2000
MOTION 0x20 GOTO 0x04EC
PAUSE 2000
; Repeat the program from 180 degrees
GOTO #180
```

PING MCU_ID [Count] [Delay]

Requests an on-line status response from a micro controller. The *Count* parameter is the number of response packets desired. The *Delay* parameter is the number of millisecond delays between replies. If no parameters are specified, only one reply is sent by the micro controller. The PING command is synchronous for the first reply meaning that the *MCLI* will wait for the first response before executing the next instruction. After the first reply the *MCLI* continues executing instructions even if not all the requested replies have been received. This is by design.

PING *MCU_ID* [Count] [Delay]			
SID	SID_MCU_ID	0x0	MSG_STATUS
0	TX_MCU_ID	CAT_EXEC	IX_STAT_PING
1			***Count***
2			***Delay*** **ms**
3			

<u>Example</u>

```
; Ping the Gateway MCU
PING 0x10                ; 1 Sync Reply
PING 0x10 0x02           ; 1 Sync and 1 async Replies; no delay
PING 0x10 0x04 0x01      ; 1 Sync and 3 async Replies; 1 ms delay between replies

; Ping a CAN Node
PING 0x20                ; 1 Sync Reply
PING 0x20 0x02           ; 1 Sync and 1 async Replies; no delay
PING 0x20 0x04 0x01      ; 1 Sync and 3 async Replies; 1 ms delay between replies

; Ping All the Micro Controllers
PING 0x00                ; Request On-Line Status Response from all nodes
```

The first reply received after a PING is the reply that ends the wait. An asynchronous reply from a previous PING (executing multiple replies) will end the waiting of any PING instruction after it. For MCU_ID not equal 0x00, the *PING* instructions are synchronous. Broadcasts to 0x00 are asynchronous after the first MCU responds.

PING Reply returns the *Firmware_Date*, *Version*, and *Reply_Sequence* countdown number. If the *Count* parameter was 16, then the each reply packet will have a *Reply_Sequence* number from 16 to 1.

SID	SID_MCU_ID	0x0	MSG_STATUS
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_STAT_PING**
1	***Year***		***Month***[hi], ***Day***[lo]
2	***Version***		***Reply_Sequence***
3	***0***		***0***

POSITION MCU_ID [*NAV* | *QEI*] [TimeOut]

Requests the current Position information from a motor controller. The *NAV* option requests the Navigable Space Coordinates; the Logical Position. The *QEI* option requests the Encoder Register Values; the Physical Position. The *POSITION* command is always synchronous so the values returned may be used by subsequent program instructions. A *TimeOut* value in milliseconds may be specified. If the *TimeOut* expires, the program is halted.

POSITION *MCU_ID* [Count] [Delay]			
SID	SID_MCU_ID	0x0	MSG_POSITION
0	TX_MCU_ID	CAT_EXEC	IX_POSI_[NAV|QEI]
1			
2			
3			

Example

```
; Synchronously Query Motor Controller 0x20 for its position
; Get the Current Navigable Space Coordinates
POSITION 0x20 NAV
; Get the Current Encoder Registers
POSITION 0x20 QEI
; Asynchronously Get All the Motor Controller Positions
POSITION 0x00 NAV
POSITION 0x00 QEI
```

For MCU_ID not equal 0x00, the *POSITION* instructions are synchronous. Broadcasts to 0x00 are asynchronous after the first MCU responds. The values returned are saved in the following UInt16 arrays:

```
MotPosi_Nav[16]    ; Navigable Space Position
MotPosi_Qei[16]    ; Quadrature Encoder Position
MotPosi_Sec[16]    ; Position Sector
MotPosi_Dir[16]    ; Motor Direction: 0 = Forward, 1 = Reverse or vice-versa
MotPosi_Odo[16]    ; Odometer (number of times crossing the Home position.
```

SID	SID_MCU_ID	0x0	MSG_POSITION
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_POSI_NAV**
1	***Nav_Position***[HI]		***Nav_Position***[LO]
2	***Direction***		***Sector***
3	***Odometer***[HI]		***Odometer***[LO]

SID	SID_MCU_ID	0x0	MSG_POSITION
0	**TX_MCU_ID**	**CAT_RPLY**	**IX_POSI_QEI**
1	***QEI_Position***[HI]		***QEI_Position***[LO]
2	***Direction***		***Sector***
3	***Odometer***[HI]		***Odometer***[LO]

MCL Interpreter

The Motion Control Language, MCL, is interpreted and its instructions are executed by an application on a PC workstation. The MCL Interpreter, MCLI, transmits the compiled MCL instruction codes the Gateway Micro Controller, GW_MCU. If the SID_MCU_ID = 0x10 or 0x00 (broadcast), the the GW_MCU will execute the instruction. If the SID_MCU_ID is not 0x10, then the GW_MCU will relay it to the CAN.

The following is a screen shot of the MCL Interpreter after it has completed a simple PING Test program. The last *Received Packet* is displayed in the upper left with the execution log displayed below it. The right side of the screen shows the program editor.

MCL Command Interpreter

File

On-Line

Received Packet

Rx Pkt # 15 — Data 20 7
SID MCU_ID f0 — Msg 1 — 11 14
TX MCU 20 — Cat 2 — Inst 1 — 0 1

```
** Program Start **
PING 0x10
-> RX# 1: 10[1], 20, 7, 11, 14, 0, 1
PING 0x10 0x02
-> RX# 2: 10[1], 20, 7, 11, 14, 0, 2
PING 0x10 0x04 0x01
-> RX# 3: 10[1], 20, 7, 11, 14, 0, 1
-> RX# 4: 10[1], 20, 7, 11, 14, 0, 4
-> RX# 5: 10[1], 20, 7, 11, 14, 0, 3
PING 0x20
-> RX# 6: 10[1], 20, 7, 11, 14, 0, 2
PING 0x20 0x02
-> RX# 7: 10[1], 20, 7, 11, 14, 0, 1
PING 0x20 0x04 0x01
-> RX# 8: 20[1], 20, 7, 11, 14, 0, 1
-> RX# 9: 20[1], 20, 7, 11, 14, 0, 1
-> RX# 10: 20[1], 20, 7, 11, 14, 0, 4
PING 0x00
-> RX# 11: 20[1], 20, 7, 11, 14, 0, 3
Done!
-> RX# 12: 20[1], 20, 7, 11, 14, 0, 2
-> RX# 13: 20[1], 20, 7, 11, 14, 0, 1
-> RX# 14: 10[1], 20, 7, 11, 14, 0, 1
-> RX# 15: 20[1], 20, 7, 11, 14, 0, 1
```

C:\MuAxMo\RCGW\MCL\PING_TEST.mcl

```
; Ping the Gateway MCU
PING 0x10           ; 1 Sync Reply
PING 0x10 0x02      ; 1 Sync + 1 Async Replies; no delay
PING 0x10 0x04 0x01 ; 1 Sync + 3 async Replies w/ 1 ms delay

; Ping a CAN Node
PING 0x20           ; 1 Sync Reply
PING 0x20 0x02      ; 1 Sync + 1 Async Replies; no delay
PING 0x20 0x04 0x01 ; 1 Sync and 3 async Replies; 1 ms delay

; Ping All the Micro Controllers
PING 0x00
```

Program Stopped: Done! — RX= 75 — TX= 35 — ERR= 0

The above ping test program example illustrates the difference between synchronous and asynchronous communications. The first PING 0x10 executes and waits for the reply (synchronous).

The second PING 0x10 0x02 command expects 2 replies; but only waits (synchronous) for the first reply and does not wait (asynchronous) for the second reply. The second reply from PING 0x10 0x02 is actually received after the PING 0x10 0x04 0x01 instruction is executed!

The completion of the program is logged as “Done!” in the log. The replies from the last 2 commands are received after the program has completed its execution. The Hold point for PING 0x00 was released by Packet #3 of the previous PING instruction. This is caused by a combination of asynchronous replies clearing the synchronous waiting and the RS232 communications buffer saving the replies.

The above PING behavior may seem confusing; but it is by design. It allows instructions to be crafted for optimal communications and to prevent the RS232 computer interface to become a bottle neck. It also permits the robot to operate autonomously without requiring constant instructions and feedback from the PC.

MOTOR CONTROL REGISTERS

There are 16 Motor Control records. Each stores an array of registers for the various motor control parameters. The records are mapped to the *MCU_ID* of each motor controller. These registers are updated by various program instructions. e.g. the *POSITION* instructions updated the *MotiPosi_[NAV, QEI, SEC, DIR, ODO]* registers.

Records 0 and 15 are "scratch pad" registers; only MCU_ID 0x1 to 0xE are motor controllers. Record 0 is used for storing data to be transmitted. Record 15 is used for received data. The MCLI writes and reads the registers as it interprets and executes instructions. The Motor Control Registers are displayed on a program tab for diagnostics.

MCL Interpreter - ver 11/18/07 0.0.1

File

On-Line | Registers

Received Packet

Rx Pkt # 66 | Data 0 29
SID MCU_ID f0 | Msg 3 | 0 4
TX MCU 20 | Cat 2 | Inst 2 | 0 0

```
** Program Start **
POSITION 0x10 NAV
-> RX# 63: 10[1], 0, 0, 1, 0, 0, 0
POSITION 0x10 QEI
-> RX# 64: 10[2], 0, 0, 1, 0, 0, 0
POSITION 0x20 NAV
-> RX# 65: 20[1], 0, 29, 0, 4, 0, 0
POSITION 0x20 QEI
-> RX# 66: 20[2], 0, 29, 0, 4, 0, 0
Done!
```

C:\MuAxMo\RCGW\MCL\POSI_TEST.mcl

```
POSITION 0x10 NAV
POSITION 0x10 QEI

POSITION 0x20 NAV
POSITION 0x20 QEI
```

Program Stopped: Done! | RX= 4 | TX= 4 | ERR= 0

MCL Interpreter - ver 11/18/07 0.0.1

File

On-Line | Registers

MCU_ID	NAV	QEI	SEC	DIR	ODO
0	0	0	0	0	0
1	0	0	0	1	0
2	41	41	4	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	0	0	0	0
15	0	0	0	0	0

Program Stopped: Done! | RX= 4 | TX= 4 | ERR= 0

GLOSSARY

The following terms are used throughout this document. These terms may or may not be in a common dictionary. The definitions of the terms below are specific to the context of this documentation.

Asynchronous - The sender of a message does not wait for an acknowledgment from the recipient

Absolute Coordinates - The QEI Register values at a position.

Controller Area Network; CAN - International standard protocol defined by ISO-11898

Motion Control Language; MCL - The programming commands used to write motion control applications

Motion Control Language Interpreter; MCLI - The PC application that executes the MCL programming commands. In our application, the MCLI communicates with the micro controllers through the gateway and the CAN.

Motion Control System; MCS - The complete robotic system from human to motor

Navigable Space - An alternate, *Logical*, coordinate system used to express motion position. For example, Navigable Space for the Index Table is 360 degrees. Navigable Space consists of *Sectors* for each unit (e.g. Index Table degree) and an Odometer count.

Odometer - A counter that increments or decrements in response to an input. For the Index Table, the Odometer counts the table revolutions across the Home position.

Ping - Poll a device across the network to see if it is on-line

PWM - Pulse Width Modulation

QEI - Quadrature Encoder Interface

Synchronous - The sender of a message waits for an acknowledgment from the recipient

UInt16 - Unsigned 16 Bit Integer