USER MANUAL

Power Brick LV ARM

Low Voltage Programmable Servo Amplifier

PBL□-A□□-0□□-□□□□□□□

December 16, 2020

Document # MN-000143

Delta Tau Data Systems, Inc.

COPYRIGHT INFORMATION

To report errors or inconsistencies, email: odt-support@omron.com.

For inquiries about the product, contact your local OMRON representative.

Trademarks

All encoder protocols and industrial networks mentioned in this manual are registered trademarks to their corresponding owners. They are only used in the purpose of product and technical description. E.g. EtherCAT® is a registered trademark of Beckhoff.

OPERATING CONDITIONS

All Delta Tau Data Systems, Inc. motion controller, accessory, and amplifier products contain static sensitive components that can be damaged by incorrect handling. When installing or handling Delta Tau Data Systems, Inc. products, avoid contact with highly insulated materials. Only qualified personnel should be allowed to handle this equipment.

In the case of industrial applications, we expect our products to be protected from hazardous or conductive materials and/or environments that could cause harm to the controller by damaging components or causing electrical shorts. When our products are used in an industrial environment, install them into an industrial electrical cabinet to protect them from excessive or corrosive moisture, abnormal ambient temperatures, and conductive materials. If Delta Tau Data Systems, Inc. products are directly exposed to hazardous or conductive materials and/or environments, we cannot guarantee their operation.

SAFETY INSTRUCTIONS

Qualified personnel must transport, assemble, install, and maintain this equipment. Properly qualified personnel are persons who are familiar with the transport, assembly, installation, and operation of equipment. The qualified personnel must know and observe the following standards and regulations:

IEC364resp.CENELEC HD 384 or DIN VDE 0100
IEC report 664 or DIN VDE 0110
National regulations for safety and accident prevention or VBG 4

Incorrect handling of products can result in injury and damage to persons and machinery. Strictly adhere to the installation instructions. Electrical safety is provided through a low-resistance earth connection. It is vital to ensure that all system components are connected to earth ground.

This product contains components that are sensitive to static electricity and can be damaged by incorrect handling. Avoid contact with high insulating materials (artificial fabrics, plastic film, etc.). Place the product on a conductive surface. Discharge any possible static electricity build-up by touching an unpainted, metal, grounded surface before touching the equipment.

Keep all covers and cabinet doors shut during operation. Be aware that during operation, the product has electrically charged components and hot surfaces. Control and power cables can carry a high voltage, even when the motor is not rotating. Never disconnect or connect the product while the power source is energized to avoid electric arcing.

Warning

A Warning identifies hazards that could result in personal injury or death. It precedes the discussion of interest.

Caution

A Caution identifies hazards that could result in equipment damage. It precedes the discussion of interest.

Note

A Note identifies information critical to the understanding or use of the equipment. It follows the discussion of interest.

MANUAL REVISION HISTORY				
REV	**DESCRIPTION**	**DATE**	**CHANGE**	**APPROVED**
A	Preliminary	10/5/2020	EH	RN
B	Released	10/12/2020	EH	RN
C	Part Number and Options Corrections Serial Encoder Sections Update Added D-SUB Mounting Warning	12/16/2020	EH	RN

Table of Contents

INTRODUCTION

The Power Brick LV is a smart servo drive package. It combines the intelligence and capability of the Power PMAC motion controller with high performance MOSFET-based drives resulting into a 4 or 8-axis compact smart drive.

The Power Brick LV is designed for "low voltage" mains input power up to 60 VDC, it supports virtually any type of feedback device and can drive directly the following types of motors:

- 3-phase AC/DC brushless servo (synchronous) -- rotary/linear
- 2-phase stepper
- 2-phase DC brush

Note

The Power Brick LV can also provide pulse and direction PFM output signals to third-party stepper drives.

The number of axes in a Power Brick LV application can be expanded through MACRO or EtherCAT.

The Power Brick LV carries up to 32 digital inputs and 16 digital outputs (I/Os) which can also be expanded through MACRO, ModBus, or EtherCAT.

The outstanding trajectory planner, built-in software PLCs (programmable in Power PMAC script and / or C language), and safety features make the Power Brick LV a fully scalable machine automation controller-drive which can be virtually integrated in any kind of motion control application.

Documentation

In conjunction with this manual, the following manuals are essential for the proper operation and use of the Power Brick LV:

- Power PMAC Software Reference Manual
- Power PMAC User Manual

These manuals are available for download, to registered members, at Delta Tau Forums.

Downloadable Power PMAC Script

Some code snippets may require the user to input specific information pertaining to their system application. They are denoted in a commentary ending with – User Input.

This manual contains downloadable code snippets in Power PMAC script. These examples can be copied and pasted into the editor area of the IDE software. Care must be taken when using pre-configured Power PMAC code, some information may need to be updated to match hardware or system specific configurations. Downloadable code found in this manual is enclosed in the following format:

```
GLOBAL MyCounter = 0                    // Arbitrary global variable, counter
GLOBAL MyCycles = 10                    // Arbitrary global variable, number of cycles –User Input

OPEN PLC ExamplePLC                     // Open PLC buffer
WHILE (MyCounter < MyCycles)            // While counter is less than number of cycles
{                                       // Start while loop
    MyCounter ++                        // Increment MyCounter by 1
}                                       // End while loop
DISABLE PLC ExamplePLC                  // Disable PLC
CLOSE                                   // Close PLC buffer
```

It is the user's responsibility to manage the application's PLCs properly. The code samples are typically enclosed in a PLC buffer with the user defined name ExamplePLC.

It is the user's responsibility to use the PLC examples presented in this manual properly, and incorporate the statement code in the application project accordingly.

RECEIVING AND UNPACKING

Delta Tau products are thoroughly tested at the factory and carefully packaged for shipment. When the Power Brick LV ARM is received, there are several things to be done immediately:

- Observe the condition of the shipping container and report any damage immediately to the commercial carrier that delivered the package.
- Remove the equipment from the shipping container and remove all packing materials. Check all shipping material for connector kits, documentation, or other small pieces of equipment. Be aware that some connector kits and other equipment pieces may be quite small and can be accidentally discarded if care is not used when unpacking the equipment. The container and packing materials may be retained for future shipment.
- Verify that the part number of the product received is the same as the part number listed on the purchase order.
- Inspect the equipment for external physical damage that may have been sustained during shipment and report any damage immediately to the commercial carrier.
- Electronic components in this product are design-hardened to reduce static sensitivity. However, use proper procedures when handling the equipment.
- If the equipment is to be stored for several weeks before use, be sure that it is stored in a location that conforms to published storage humidity and temperature specifications.

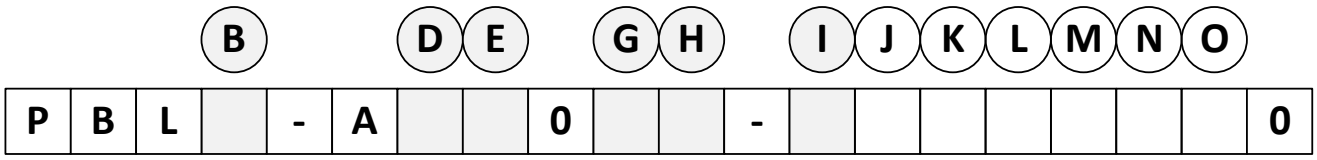
Use of Equipment

The following restrictions will ensure the proper use of the Power Brick LV:

- The components built into electrical equipment or machines can be used only as integral components of such equipment.
- The Power Brick LV must not be operated on power supply networks without a ground or with an asymmetrical ground.
- If the Power Brick LV is used in residential areas, or in business or commercial premises, implement additional filtering measures.
- The Power Brick LV may be operated only in a closed switchgear cabinet, taking into account the ambient conditions defined in the environmental specifications.

SPECIFICATIONS

Part Number Designation



Option B	
4:	4-Axis
8:	8-Axis

Option D	
A:	1GB RAM, 4GB Flash
E:	2 GB RAM, 4GB Flash

Option E	
0:	No EtherCAT®
1:	I/O only
2:	I/O + 4 Servo Axis
3:	I/O + 8 Servo Axis
5:	I/O + 16 Servo Axis
9:	I/O + 32 Servo Axis

Option I	
0:	-
1:	MACRO

Option GH			
	Axis 1-4	**Axis 5-8**	**No. Enc.**
50:	5/15A	-	4
5A:	5/15A	-	8
70:	1/3A	-	4
7A:	1/3A	-	8
80:	0.25/0.75A	-	4
8A:	0.25/0.75A	-	8
55:	5/15A	5/15A	8
77:	1/3A	1/3A	8
88:	0.25/0.75A	0.25/0.75A	8
57:	5/15A	1/3A	8
58:	5/15A	0.25/0.75A	8
78:	1/3A	0.25/0.75A	8

			B			D	E		G	H		I	J	K	L	M	N	O	
P	B	L		-	A			0			-								0

Option J	
0:	16/8 Digital I/O
1:	32/16 Digital I/O*1
A:	PROFIBUS-DP Master
B:	PROFIBUS-DP Slave
C:	DeviceNet Master
E:	DeviceNet Slave
F:	CANopen Master
G:	CANopen Slave
H:	CC-Link Slave
J:	EtherCAT Slave
K:	Ethernet/IP Scanner
L:	Ethernet/IP Adapter
M:	Open Modbus / TCP
N:	PROFINET IO RT Controller
P:	PROFINET IO RT Device

Option O				
	True DAC	Filtered PWM	Analog Inputs	GP Relays
0:	-	-	-	-
1:	-	4	8	4
2:[*1]	-	8	4	8
4:[*1]	4	-	4	4
5:[*1]	4	4	4	8

Option K	
	Axis 1-4
0:	-
A:	ACI*2
R:	Resolver
S:	Sinusoidal

Option M*4	
	Axis 1-4
0:	-
2:	SSI
3:	EnDat
4:	HiperFace
6:	Yaskawa III/V
7:	Tamagawa
8:	Panasonic
9:	Mitutoyo
B:	BiSS B/C
C:	Matsushita
D:	Mitsubishi
E:	Omron 1S
F:	TBPC*3
G:	XY2-100

Option L*1	
	Axis 5-8
0:	-
A:	ACI*2
R:	Resolver
S:	Sinusoidal

Option N*1*4	
	Axis 5-8
0:	-
2:	SSI
3:	EnDat
4:	HiperFace
6:	Yaskawa III/V
7:	Tamagawa
8:	Panasonic
9:	Mitutoyo
B:	BiSS B/C
C:	Matsushita
D:	Mitsubishi
E:	Omron 1S
F:	TBPC*3
G:	XY2-100

*1. Only available with 8 encoders option in GH
*2. ACI: Auto-Correcting Interpolator
*3. TBPC: Table-Based Position Compare
*4. ACC-84B Options

Power Brick LV Configuration

The Power Brick LV comes standard with a powerful set of hardware and software capabilities, plus a full set of options.

Standard Configuration

CPU	1.0 GHz Dual-Core ARM
Memory	1 GB DDRAM3, 1 GB Flash.
Communication Ports	2 x Gbs Ethernet port for host communication. USB 2.0 Host port. USB 2.0 Mass Storage and-Serial UART Diagnostic Port
Digital I/O	16 x Inputs, fully protected at 12 – 24 V sourcing or sinking (user wiring). 8 x Outputs, fully protected at 12 – 24 V sourcing or sinking (user wiring).
Servo Interface	Four channels servo interface, each including: Quadrature encoder (differential, with index) interface. UVW digital hall sensor interface. Serial encoder interface (software configurable): o SSI o EnDat 2.1 / 2.2 (2.1-compatible features only) with delay compensation o Hiperface o Yaskawa Sigma I / II / III / V (no position reset or fault clear) o Tamagawa FA-Coder (no servo clock output) o Panasonic (no servo clock output) o Mitutoyo o Kawasaki Pulse & direction output. Position compare (EQU) output (5 V TTL). Input flags (home, + limit, – limit, user) at 5 – 24 V. Motor thermal input (PTC).
Amplifier Output	4 amplifier axes, each at 5/15A,
Amplifier Safety & Features	Hardware I2T thermal fault detection. Short circuit detection. PWM frequency out-of-range detection. Watchdog output (normally closed / open). Abort Input (category 2 stop). STO Input (category 0 stop).

Options

Memory	2 GB DDRAM3, 4 GB Flash.	
Digital I/O	Additional 16 x Inputs, fully protected at 12 – 24 V sourcing or sinking (user wiring). Additional 8 x Outputs, fully protected at 12 – 24 V sourcing or sinking (user wiring).	
Analog I/O	4 or 8 x 16-bit analog inputs. 4 or 8 x 14-bit filtered PWM analog outputs (±10 V). 4 x 16-bit true DAC analog outputs (±10 V). 4 or 8 x Amp enable outputs (to 3rd party drives). 4 or 8 x Amp fault inputs (from 3rd party drives).	
Servo Interface	Four additional servo channels with optional: Sinusoidal encoder interface (x16384). Auto-Correcting Interpolator ACI sinusoidal encoder interface (x65536) Resolver encoder interface. ACC-84B protocols: ○ SSI (no additional capability over Gate3 built-in interface) ○ EnDat 2.2 with additional information, no delay compensation ○ Hiperface (no additional capability over Gate3 built-in interface) ○ Yaskawa Sigma II/III/V with position reset and fault clear ○ Tamagawa FA-Coder with servo clock output ○ Panasonic (no additional capability over Gate3 built-in interface) ○ Mitutoyo (no additional capability over Gate3 built-in interface) ○ BiSS-B/C ○ Matsushita (Nikon D) ○ Mitsubishi ○ Omron1S ○ Table Based Position Compare Provided by ACC-84B ○ XY2-100 Provided by ACC-84B	
Amplifier Output	4 additional amplifier axes. Sets of 4-axes can be: 0.25 / 0.75 A or 1 / 3 A or 5 / 15 A	
MACRO Interface	16 Servo, 12 I/O nodes interface. 32 Servo, 24 I/O nodes interface.	
EtherCAT Interface	EtherCAT I/O only. 4 / 8 / 16 / 32 Servo axes plus I/O.	
Fieldbus	EtherNet / IP Scanner / Master. EtherNet / IP Adapter / Slave. Open Modbus / TCP. PROFINET IO RT Controller. PROFINET IO RT Device. CANopen Master. CANopen Slave.	PROFIBUS-DP Master. PROFIBUS-DP Slave. DeviceNet Master. DeviceNet Slave. CC-Link Slave. EtherCAT Slave. Modbus.

Configuration Notes

- Quadrature encoders can always be wired in and processed regardless of the feedback options fitted.

- The following serial encoder protocols are built into (standard) the Power Brick LV – Gate3:

HiperFace	Kawasaki	Tamagawa
SSI	EnDat 2.1 / 2.2	Mitutoyo
Panasonic	Yaskawa II / III / V	

  Additionally, any of the listed optional protocols can be ordered (in sets of 4 channels: 1 – 4 or 5 – 8). These are processed on what is known as the ACC–84B (piggy back inside the Power Brick LV).

  Some protocols may overlap between the Gate3 and ACC–84B. Users may need new, updated protocols, or additional serial data information which may not be available with the standard Gate3 protocol implementation.

- With the optional ACC-84B installed, a given channel can be configured (in software) to use either one the Gate3 serial encoder protocols or one of the ACC-84B protocols.

- If a serial encoder is used on a given channel, it is also possible to wire in on the same connector and process simultaneously a quadrature/sinusoidal/resolver encoder.

Note that, pins #5, 6, 13, and 14 of the encoder feedback connectors (X1 – X8) share multiple functions: only one of these functions (per channel) can be used – configured in software – at one time:

- Hall sensor inputs (default configuration).
- Pulse and direction PFM output signals (enable using **PowerBrick[].Chan[].OutFlagD**).
- Serial encoder inputs (enable using **PowerBrick[].SerialEncEna**).
- Serial encoder inputs (enable using bit 10 of **ACC84B[].SerialEncCmd** with ACC-84B).
- Quadrature encoder inputs (serial encoder input must be disabled).
- Alternate Sinusoidal encoder inputs (with sinusoidal encoder option).

Note

Each channel is independent of the other channels and can have its own use for these pins.

Environmental Specifications

Specification	Description	Range
Ambient operating Temperature EN50178 Class 3K3 – IEC721-3-3	Minimum operating temperature	0 °C (32 °F)
	Maximum operating temperature	45 °C (113 °F)
Storage Temperature Range EN 50178 Class 1K4 – IEC721-3-1/2	Minimum Storage temperature	-25 °C (-13 °F)
	Maximum Storage temperature	70 °C (158 °F)
Humidity Characteristics with NO condensation and NO formation of ice IEC721-3-3	Minimum Relative Humidity	5% HU
	Maximum Relative Humidity up to 35°C (95°F)	95% HU
	Maximum Relative Humidity from 35°C up to 50°C (122°F)	85% HU
De-rating for Altitude	0 ~ 1000 m (0 ~ 3300 ft)	No de-rating
	1000 ~ 3000 m (3300 ~ 9840 ft)	-0.01%/m
	3000 ~ 4000 m (9840 ~ 13000 ft)	-0.02%/m
Environment ISA 71-04	Degree 2 environments	
Atmospheric Pressure EN50178 class 2K3	70 kPa to 106 kPa	
Air Flow Clearances	3" (76.2 mm) above and below unit for air flow	
Cooling	Natural convection and external fan	
Standard IP Protection	IP20 IP 55 can be evaluated for custom applications	

Electrical Specifications

Output Current (per Axis)	**Continuous**	**Peak**
Possible Configurations	0.25 A_{rms}	0.75 A_{rms}
	1 A_{rms}	3 A_{rms}
	5 A_{rms}	15 A_{rms}
Time at peak current	1 second	

Item	**0.25/1A**	**1/3A**	**5/15A**
Max ADC Reading	1.6925 A_{peak}	6.770 A_{peak}	33.85 A_{peak}
Logic power supply input	24 VDC ± 5%		
Logic power supply current	5 A_{rms}		
Recommended PWM Frequency	40 KHz	40 KHz	20 KHz
Max. PWM Frequency	100 KHz	100 KHz	30 KHz

Main Power Supply Input	**4-Axis**			**8-Axis**		
	0.25/0.75A	**1/3A**	**5/15A**	**0.25/0.75A**	**1/3A**	**5/15A**
Maximum Voltage	60 VDC					
Full Load Current F.L.A.*	2 A	5 A	21 A	3 A	9 A	25 A

* At 60VDC main power input, and continuous (e.g. 0.25A, 1A, or 5A) current output per axis

Maximum Power Output*	**4-Axis**			**8-Axis**		
	0.25/0.75A	**1/3A**	**5/15A**	**0.25/0.75A**	**1/3A**	**5/15A**
Power Output (total)	56 W	228 W	1060 W	112 W	456 W	1325 W
Maximum No. of Axis	4	4	4	8	8	5
Power Output (Per axis)	14 W	57 W	265 W	14 W	57 W	265 W

* At 60VDC main power input, and continuous (e.g. 0.25A, 1A, or 5A) current output per axis

MOUNTING

The location of the Power Brick LV is important. Installation should be in an area that is protected from direct sunlight, corrosives, harmful gases or liquids, dust, metallic particles, and other contaminants. Exposure to these can reduce the operating life and degrade performance of the drive.

Several other factors should be carefully evaluated when selecting a location for installation:

- For effective cooling and maintenance, the Power Brick LV should be mounted on a smooth, non-flammable vertical surface.
- At least 76 mm (3 inches) top and bottom clearance must be provided for air flow. At least 10 mm (0.4 inches) clearance is required between units (each side).
- Temperature, humidity and Vibration specifications should also be taken in account.

Caution

Unit must be installed in an enclosure that meets the environmental IP rating of the end product (ventilation or cooling may be necessary to prevent enclosure ambient from exceeding 45° C [113° F]).

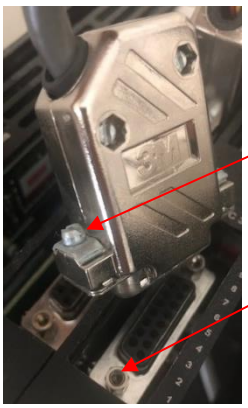
The Power Brick LV can be mounted with a traditional 3-hole panel mount, two U shape/notches on the bottom and one pear shaped hole on top.

If multiple Power Brick LVs are used, they can be mounted side-by-side, leaving at least a 122 mm clearance between drives. This means a 122 mm center-to-center distance (0.4 inches). It is extremely important that the airflow is not obstructed by the placement of conduit tracks or other devices in the enclosure.
If the drive is mounted to a back panel, the back panel should be unpainted and electrically conductive to allow for reduced electrical noise interference. The back panel should be machined to accept the mounting bolt pattern of the drive.

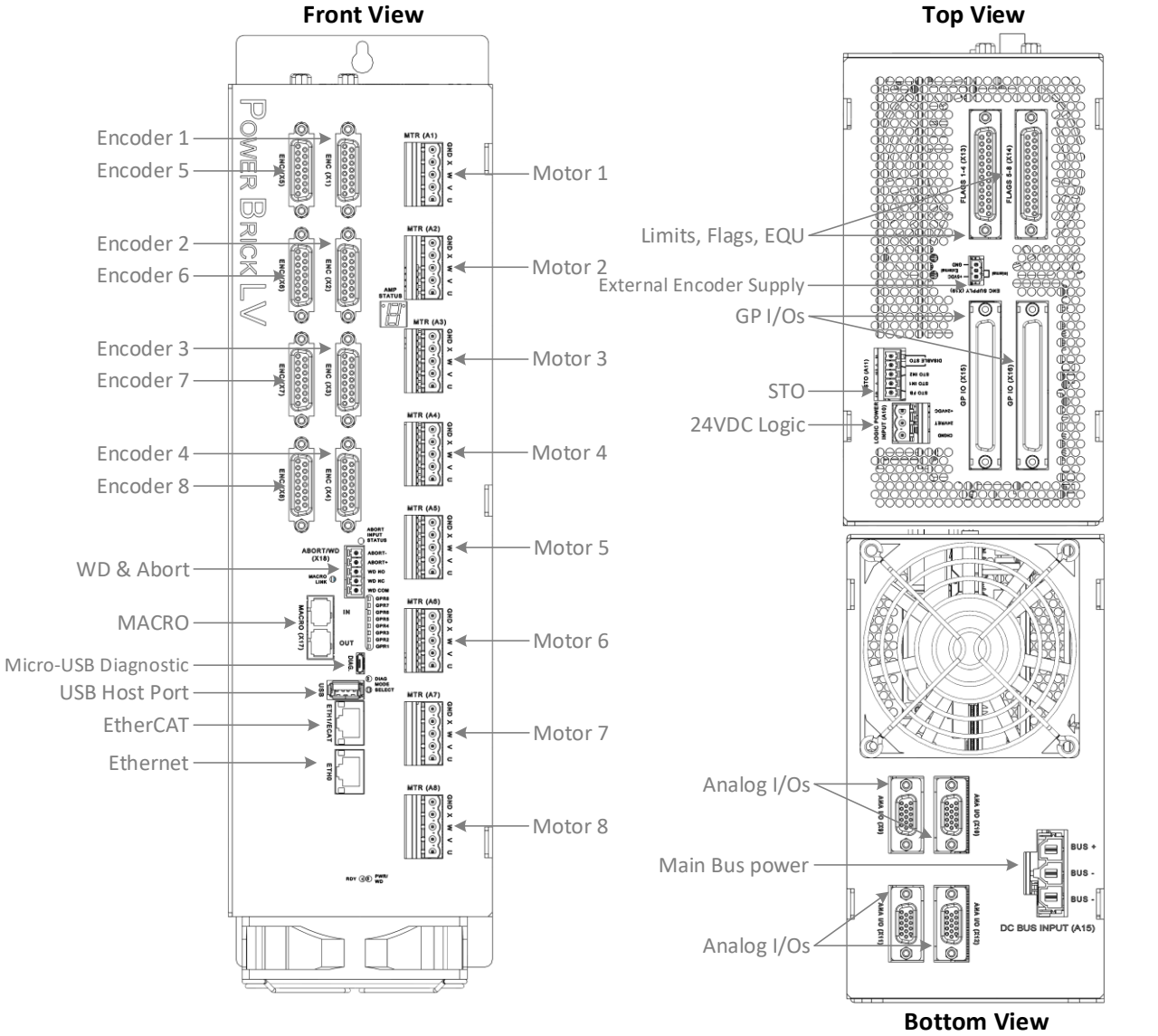
The Power Brick LV can be mounted to the back panel using three M4 screws and internal-tooth lock washers. It is important that the teeth break through any anodization on the drive's mounting gears to provide a good electrically conductive path in as many places as possible. Mount the drive on the back panel so there is airflow at both the top and bottom areas of the drive (at least three inches).

Caution

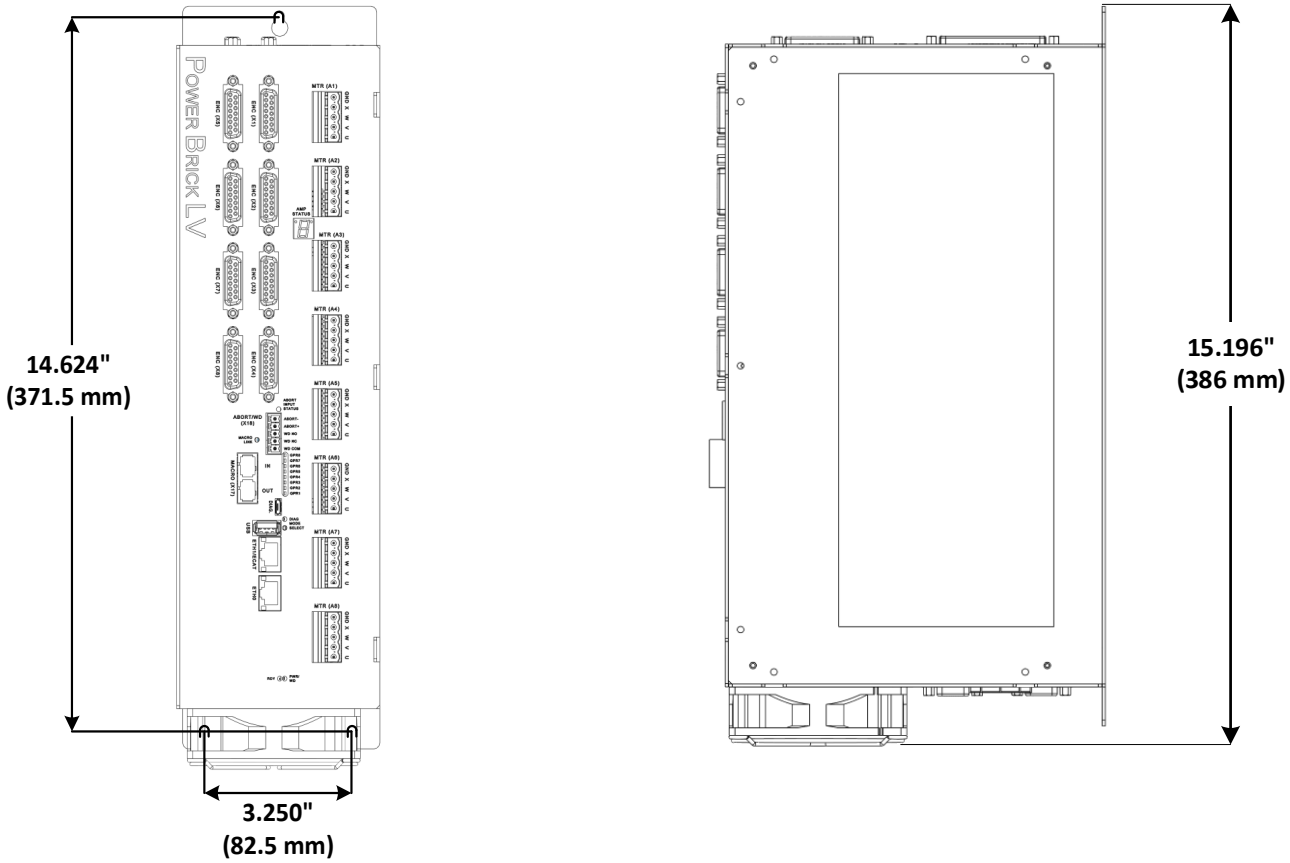
This product contains D-SUB style connectors. Do not overtighten any screws on mating connectors, and when possible, only tighten by hand. Overtightening, such as with manual or electric tools, may lead to the mating nut detaching when subsequently unscrewed, which may result in damage to the product, other electronics, and/or injury.

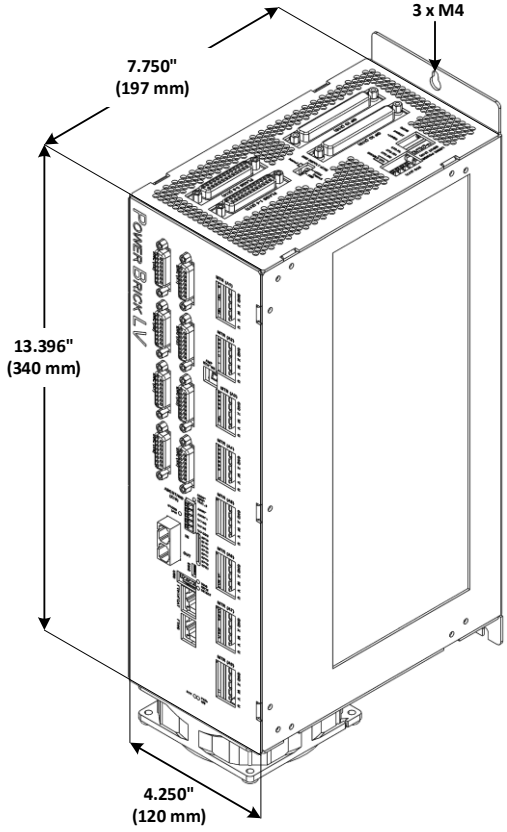


Connector Locations



CAD Drawing


14.624"
(371.5 mm)
3.250"
(82.5 mm)
15.196"
(386 mm)


3 x M4
7.750"
(197 mm)
13.396"
(340 mm)
4.250"
(120 mm)

CONNECTIONS AND BASIC SETTINGS

Warning

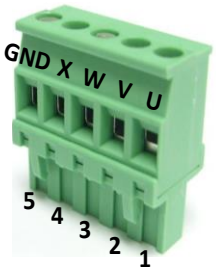
Installation of electrical control equipment is subject to many regulations including national, state, local, and industry guidelines and rules. General recommendations can be stated but it is important that the installation be carried out in accordance with all regulations pertaining to the installation.

Motor Connection (Amp1-Amp8)

These connections are used to wire the amplifier-motor output. The Power Brick LV offers three possible power configurations (per set of 4 axes, 1 – 4 or 5 – 8):

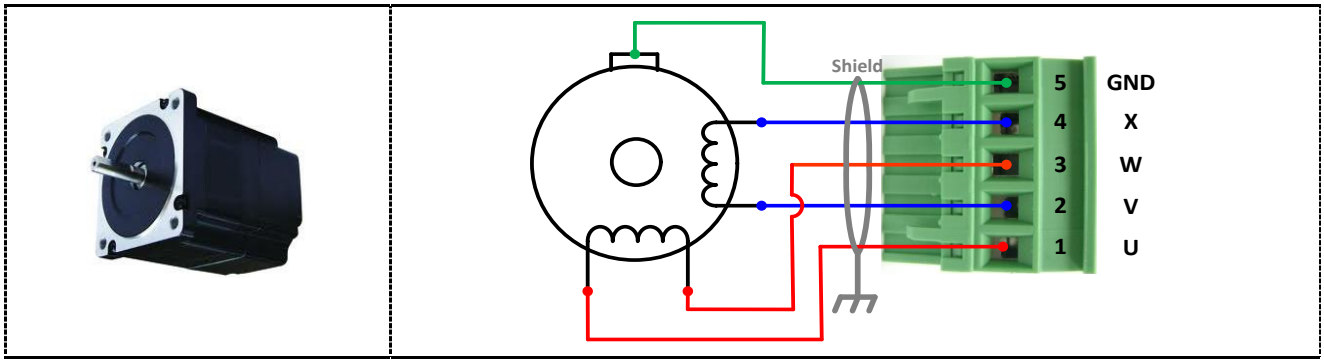
Nominal RMS Current	Peak RMS Current	Connector	Notes
0.25 A	0.75 A		Left hand side indicator
1 A	3 A		Right hand side indicator
5 A	15 A		No indicator

- For Stepper motors, use U and W at one coil, V and X at the other coil.
- For DC brushless motors (servo) use U, V and W. Leave X floating.
- For DC Brush motors, use U and W. Leave V and X floating.

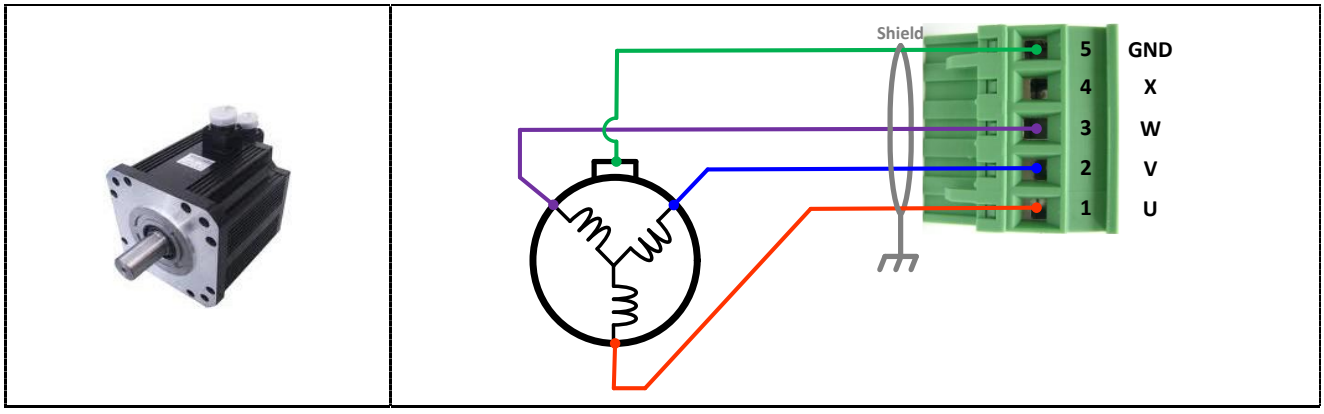
Mating Connector 5-pin Phoenix Terminal Block: Phoenix Contact mating connector part # 1757048 Delta Tau mating connector part # 016-090A05-08P	

Pin#	Symbol		Function	Description
1	Phase 1	U	Output	Motor Output
2	Phase 2	V	Output	Motor Output
3	Phase 3	W	Output	Motor Output
4	Phase 4	X	Output	Motor Output
5	GND		Common	

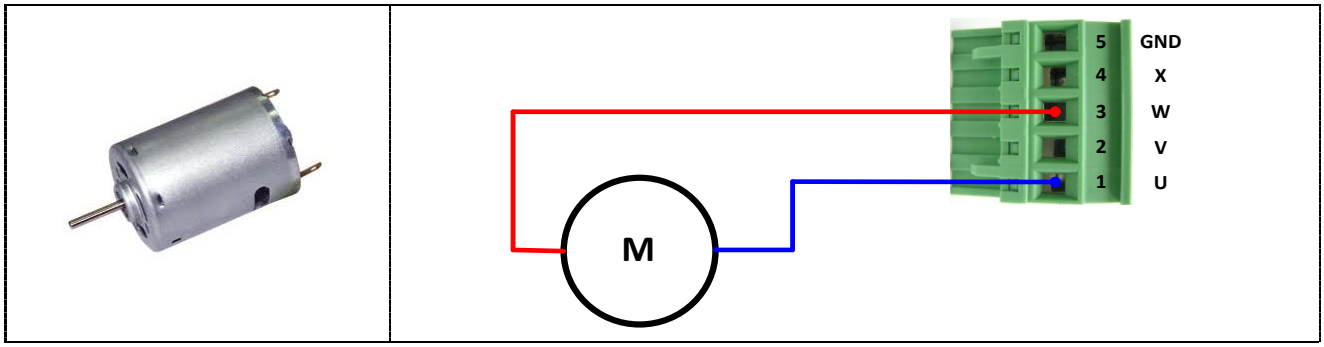
Stepper Motor Wiring



Brushless (Servo) Motor wiring



Brush Motor Wiring



Note

The motor's frame drain wire and the motor cable shield should be tied together to minimize noise disturbances.

Note

Color code may differ from one motor manufacturer to another. Review the motor documentation carefully before making this connection.

Logic Power Supply (A10)

A10 is used to bring in the 24-Volt DC supply powering up the logic portion of the Power Brick LV. This power can remain on regardless of the main DC bus power, allowing the signal electronics to be active while the main motor power is passive.

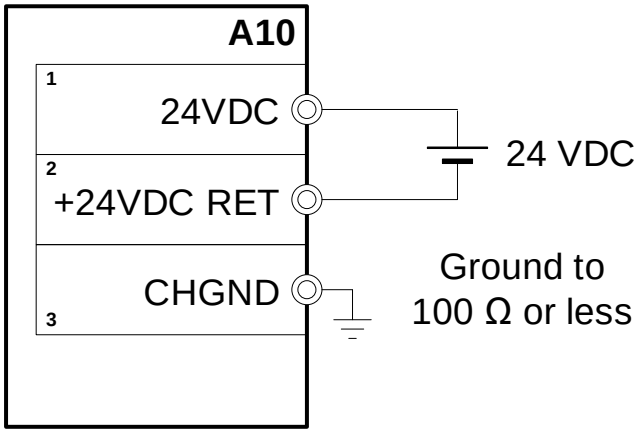
Caution

The 24V logic power must always be applied before applying main DC bus power.

The 24-Volt (±5%) power supply unit must be capable of providing 5 amperes per Power Brick LV. If multiple drives are sharing the same 24-Volt power supply, it is highly recommended to wire each drive back to the power supply terminals separately.

This connection can be made using a 22 AWG wire directly from a protected power supply.

A10: 3-pin Female **Mating: 3-pin Male**				
Pin #	**Symbol**	**Function**	**Description**	**Notes**
1	+24 VDC	Input	Logic power input +	+24 VDC (±5 %)
2	+24 VDC RET	Common	Logic power return -	Connect to Power Supply Return
3	CHGND	Ground	Chassis ground	Connect to Protection Earth
Phoenix Contact mating connector part# # 1777293				



Safe Torque OFF (A11)

A11 is used to wire the Safe Torque OFF (STO) function which allows the complete "hardware" disconnection of the power amplifiers from the motors. This mechanism prevents unintentional "movement of" or torque output to the motors in accordance with IEC/EN safety standards.

The wiring of A11 dictates whether the STO function is armed / disabled, or triggered / unprompted.

A11: 5-pin Female **Mating: 5-pin Male**			
Pin #	**Symbol**	**Function**	**Description**
1	STO FB	Output	STO Feedback
2	STO IN 1	Input	STO Input #1
3	STO IN 2	Input	STO Input #2
4	STO DISABLE	-	STO disable
5	STO DISABLE RTN	-	STO disable return
Phoenix Contact Mating Connector Part #: 1850699			

Dynamic Braking

Dynamic braking forces the motor(s) to stop from coasting freely when killed. This is done inside the Power Brick LV by tying the motor leads together. The following table summarizes the various dynamic braking scenarios (**when an axis is killed**) with respect to the STO function:

Safe Torque Off (STO)	**Dynamic Braking**
Disabled (not wired)	✓
Enabled (wired) but Not Triggered	✓
Enabled (wired) and Triggered	✗

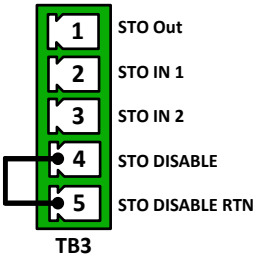
Note

When the STO is triggered, dynamic braking is not applied.

Disabling the STO

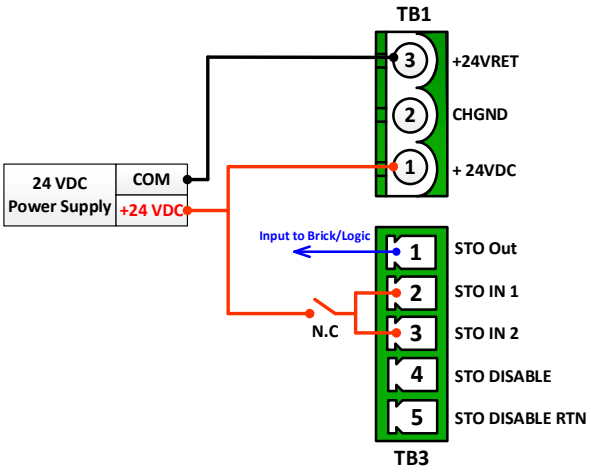
The STO can be fully disabled by tying STO disable (pin #4) to STO Disable RTN (pin #5).

Pins 1, 2 and 3 have no practical use in this mode, and should be left floating.

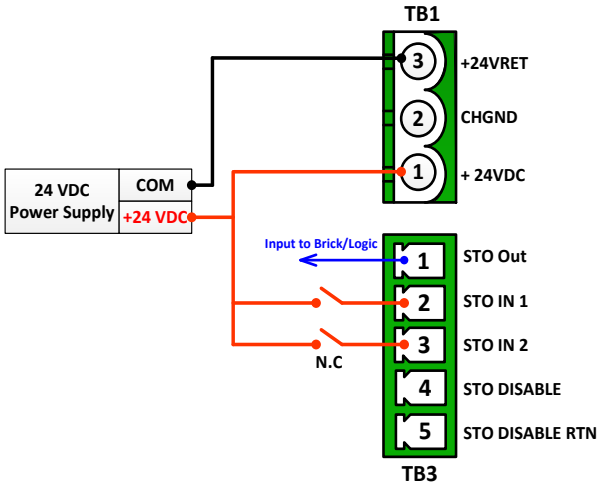


Wiring and Using the STO

➢ **SINGLE STO TRIGGER**



➢ **DUAL STO TRIGGER(S)**



- In normal mode operation, the STO relay(s) must be normally closed. +24 VDC must be applied to both STO inputs (pins #2, #3) to allow power to the motors.
- The STO is triggered and power is disconnected from the motors if the +24 V is disconnected from either STO inputs (pins #2, #3).
- The STO Out (pin #1) is a voltage status output rated to 24 VDC ±10% at a max of 125 mA. It reflects the status of the STO function:
  - (24 V) in normal mode operation (+24 VDC connected to both STO inputs)
  - (0 V) in triggered mode (+24 VDC disconnected from either STO inputs)
- Certain safety standards require dual protection, thus mandating the use of two STO input triggers.
- The STO relay(s) can be wired in series with the E-Stop circuitry which typically disconnects the main bus power from the system.

Summary of operation and status (STO Out):

+24 VDC	STO State	STO Out
Applied to both STO Inputs	Not Triggered (normal mode operation)	24 V
Disconnected from either STO inputs	Triggered	0 V

STO Feedback

The STO FB signal can be read in by a sinking input. If logic and IO have separate power supplies, this requires tying together their grounds. This input will be true when PMAC is operating normally and false when the STO disconnects power.

Recovering from the STO

The Power Brick LV does not exhibit an amplifier fault in the motor status when the STO is triggered, it is strongly advised to issue a kill to all active motors as soon as the STO is triggered. This can be done in a background PLC.

The STO disconnects power completely from the power transistors' gate drivers which briefly appears to the controller as if the current readings from the ADC sensors are saturated thus charging and tripping an I2T fault in PMAC.

The example PLC below, assuming the STO feedback FB is wired to input#1 of the Power Brick LV, kills the motor immediately and discharges I2T allowing quick recovery and regaining motor control.

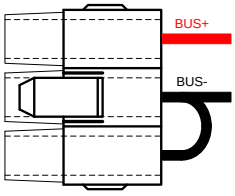
```
OPEN PLC StoResetPLC
LOCAL Mtr1PrevI2TSet

// STO TRIGGERED?
IF (Input1)
{
   // I2T CHARGED?
   IF (Motor[1].I2tSum > 0)
   {
       KILL 1
       Mtr1PrevI2TSet = Motor[1].I2TSet
       DO{Motor[1].I2tSet = 0} WHILE(Motor[1].I2tSum > 0)
       Motor[1].I2TSet = Mtr1PrevI2TSet
   }
   WHILE(Input1){}
}
CLOSE
```

Main Bus Power Supply (J1)

J1 is used to bring in the main DC bus (motor) power. The mating connecter is a Molex male 10.00 mm (.393") Pitch Mini-Fit Sr.™ Receptacle Housing, Single Row, 3 Circuits.

Pin #	Symbol	Function	Description	Notes
1	BUS+	Input	Bus power input Bus+	+12 – 60 VDC
2	BUS-	Common	Bus power return Bus-	Return Line
3	BUS-	Common	Bus power return Bus-	Return Line

Molex mating connector part# **0428160312**
Delta Tau mating connector part # 016-090003-049



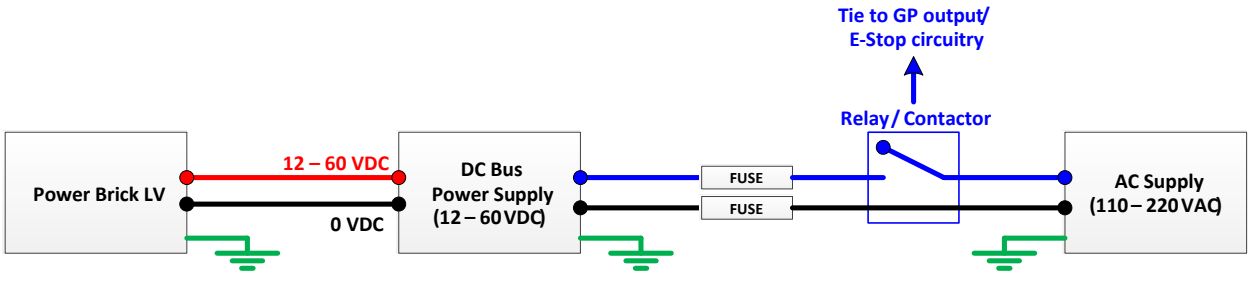
This connection can be made using the following wire gauge and fusing:

Model	Fuse (FRN/LPN)	Wire Gauge
4-Axis	15 A	12 AWG
8-Axis	25 A	10 AWG

Caution

The main DC bus power should only be toggled from the AC side of the providing power supply.

Live plugging (hotwiring) DC voltage directly into (J1) the Power Brick LV may cause unwanted effects (sparking) and could damage the operating electronics. The DC bus power supply should only be turned on/off from the AC side of the supplying circuitry.



Optionally, the relay/contactor can be tied to a general purpose output in the Power Brick LV and or incorporated into the E-Stop circuitry.

Power On/Off Sequence

Caution

The main DC bus power should NEVER be brought into the Power Brick LV if the 24 VDC logic power is NOT applied.

Caution

Make sure that no motor commands (e.g. phasing, jogging) are being executed at the time of applying main DC bus power.

Powering up the Power Brick LV must obey the following sequence:

1. Apply 24 VDC logic power (A10). The STO status can be set simultaneously. If it is in use, it can be either armed or triggered (user configurable).
2. Wait for the Power PMAC to boot up.
3. Delay 250 milliseconds then issue a **BrickLV.Reset = 1** (Power On Reset PLC).
4. Wait for **BrickLV.Reset** to get set to **0** (Power On Reset PLC).
5. Ok to apply main bus power (J1) (e.g. Release E-Stop).
6. Ok to energize motors (e.g. Press Reset).

Powering down the Power Brick LV must obey the following sequence:

1. Disconnect main bus power (J1). The STO status can be set simultaneously. If it is in use, it can be either armed or triggered (user configurable) (e.g. Engage E-Stop).
2. Wait about ~ 1 second.
3. Disconnect 24 VDC logic power (A10).

Note

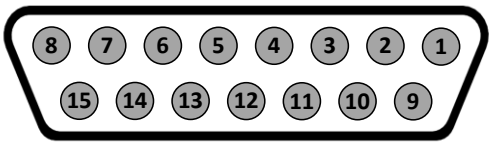
Killing all motors upon engaging the E-Stop is highly recommended. This could be triggered by a general purpose input which is typically tied to the E-Stop button/circuit.

Encoder Connection (X1-X8)

This section describes the wiring of various encoder protocols, and their basic software configuration.

Digital Quadrature

The Power Brick LV accepts digital quadrature (also known as incremental) encoder signals by default. It provides up to four counts per square cycle, and extends it using hardware-computed (ASIC) 1/T.

X1-X8: D-sub DA-15F **Mating: D-sub DA-15M**				
Pin#	**Symbol**	**Function**	**Primary Use**	**Alternate Use**
1	CHA +	Input	Encoder A +	
2	CHB +	Input	Encoder B +	
3	CHC +	Input	Index C +	
4	**ENCPWR**	**Output**	**Encoder Power 5 VDC (max 250 mA per channel)**	
5	CHU / DIR +	In / Out	Halls U	Direction Out + / Serial Data–
6	CHW / PUL +	In / Out	Halls W	Pulse Out + / Serial Clock–
7	2.5V	Output	2.5 VDC Reference power	
8	PTC	Input	Motor Thermal Input	
9	CHA –	Input	Encoder A –	
10	CHB –	Input	Encoder B –	
11	CHC –	Input	Index C –	
12	**GND**	**Common**	**Common ground**	
13	CHV / DIR –	In / Out	Halls V	Direction Out – / Serial Clock+
14	CHT / PUL –	In / Out	Halls T	Pulse Out – / Serial Data+
15				

Note

Quadrature encoders can be wired in and processed regardless of the encoder feedback option(s) the Power Brick LV is ordered with.

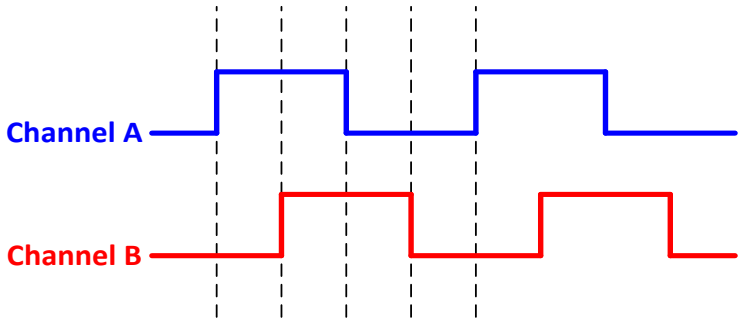
Caution

The +5 VDC encoder power is limited to ~250 mA per channel. For encoders requiring more current, the +5 VDC power can be alternately brought in externally through the +5 VDC ENC connector.

Caution

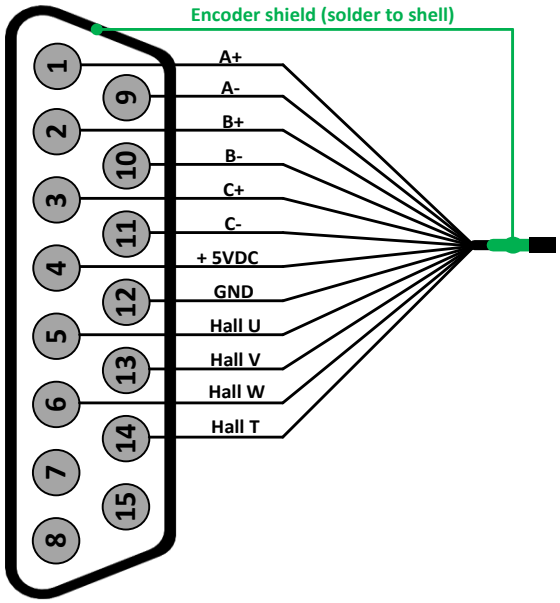
Encoders requiring a voltage level other than +5 VDC (higher or lower) should be powered up using an external power supply directly into the encoder.

Quadrature encoders provide two digital signals to determine the position of the motor. These signals are typically 5 VDC TTL/CMOS level. Each nominally with 50% duty cycle and 1/4 cycle apart. This format provides four distinct states per cycle of the signal, or per line of the encoder. The phase difference of the two signals permits the decoding electronics to discern the direction of travel, which would not be possible with a single signal.

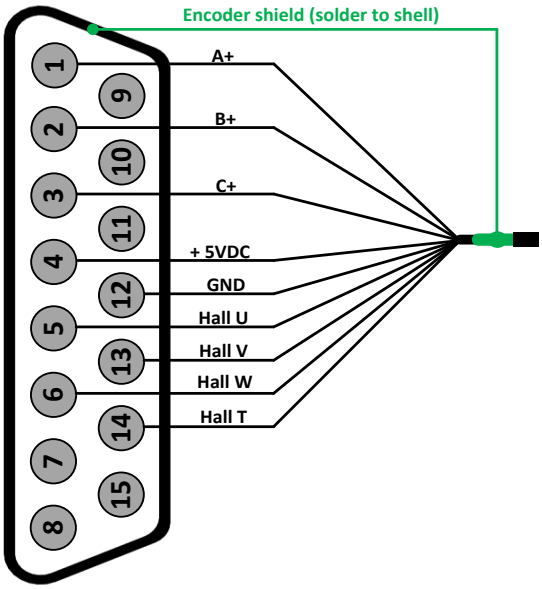


Quadrature encoders can be wired either in a differential or single-ended manner. Differential signals can enhance noise immunity by providing common mode noise rejection. Modern design standards virtually mandate their use in industrial systems.

➢ **DIFFERENTIAL**



➢ ➢ **SINGLE-ENDED**



Note

In single-ended mode, leave the negative pins floating. They are terminated internally.

Configuring Quadrature Encoders

The Power Brick LV firmware is configured to process quadrature incremental encoders by default. This type of encoders is processed as a single 32-bit word in the Encoder Conversion Table (ECT). The 1/T extension is done in the DSPGate3 hardware. Starting from factory default settings, activating the channel is sufficient to display counts in the position window when the motor / encoder shaft is moved by hand.

Default Encoder Conversion Table for quadrature incremental encoders:

```
EncTable[1].type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].ServoCapt.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 0
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1 / 256
```

Note

The hardware 1/T extension produces 8 bits of fractional data, thus the (1 / 256) 0.00390625 scale factor.

The settings below are sufficient to view motor position in the position window, in counts.

```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

Ch. #	Source Address	Ch. #	Source Address
1	PowerBrick[0].Chan[0].ServoCapt.a	5	PowerBrick[1].Chan[0].ServoCapt.a
2	PowerBrick[0].Chan[1].ServoCapt.a	6	PowerBrick[1].Chan[1].ServoCapt.a
3	PowerBrick[0].Chan[2].ServoCapt.a	7	PowerBrick[1].Chan[2].ServoCapt.a
4	PowerBrick[0].Chan[3].ServoCapt.a	8	PowerBrick[1].Chan[3].ServoCapt.a

Quadrature Counts per line

A quadrature encoder line is equivalent to 4 counts. For example, a 2,000–line rotary encoder should result in 8,000 counts per revolution (before any gearing or coupling).

Quadrature Encoder Count Error

With quadrature encoders, the Power Brick LV has the capability of trapping encoder count (loss) errors. This is described in detail in the Encoder Count Error section of this manual.

Quadrature Encoder Loss Detection

Warning

Loss of the feedback sensor signal is potentially a very dangerous condition in closed-loop control, because the servo loop no longer has any idea what the true physical position of the motor is – usually it thinks it is "stuck" – and it can react wildly, often causing a runaway condition.

With quadrature encoders, the Power Brick LV has the capability of detecting the loss of an encoder signal. This is described in detail in the Encoder Loss Detection section of this manual.

Note

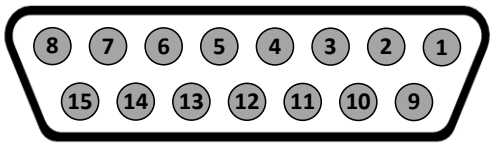
Note the distinction between the encoder count error, which reports loss of counts due to bad transitions of the quadrature signals, and encoder loss, which indicates that one or more quadrature signals are completely missing.

Analog Standard & ACI Sinusoidal

The Power Brick LV can process analog sinusoidal encoders (up to 1.2 $V_{peak-peak}$) and provide high resolution position data used in the servo loop. It is fitted with the standard or Auto Correcting Interpolator ACI if options K and or L of the part number contain S or A respectively.

- The standard option interpolation is x16384
- The ACI option interpolation is x65536 with automatic correction of sinusoidal waveform signals bias, phase, and harmonic suppression.

														K	L				
P	B	L		-	A			0			-								0

X1-X8: D-sub DA-15F **Mating: D-sub DA-15M**					
Pin#	**Symbol**	**Function**	**Primary Use**	**Alternate Use**	
1	SIN +	Input	Sine +		
2	COS +	Input	Cosine +		
3	CHC +	Input	Index C +		
4	**ENCPWR**	**Output**	**Encoder Power 5 VDC (max 250 mA per channel)**		
5	CHU / DIR +	In / Out	Halls U	Direction Out +	Serial Data – / AltSin +
6	CHW / PUL +	In / Out	Halls W	Step Out +	Serial Clock – / AltCos +
7	2.5V	Output	2.5 VDC Reference power		
8	PTC	Input	Motor Thermal Input		
9	SIN –	Input	Sine –		
10	COS –	Input	Cosine –		
11	CHC –	Input	Index C –		
12	**GND**	**Common**	**Common ground**		
13	CHV / DIR –	In / Out	Halls V	Direction Out –	Serial Clock + / AltSin –
14	CHT / PUL –	In / Out	Halls T	Step Out –	Serial Data + / AltCos –
15					

Caution

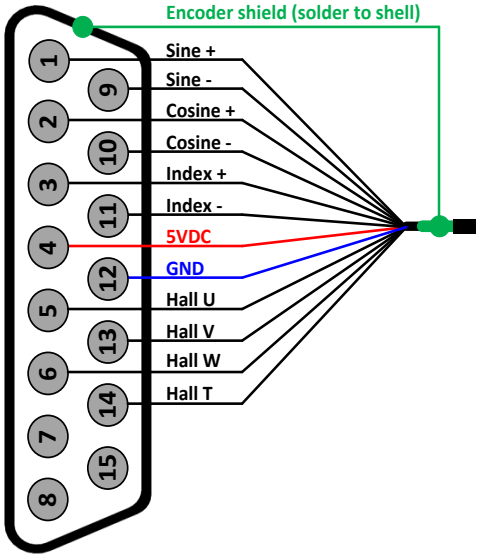
The +5 VDC encoder power is limited to ~250 mA per channel. For encoders requiring more current, the +5 VDC power can be alternately brought in externally through the +5 V ENC connector.

Caution

Encoders requiring a voltage level other than +5 VDC (higher or lower) should be powered directly from an external power supply.

The Power Brick LV can accept "sine" and "cosine" signals (90° out of phase with each other), of 1-volt (peak-to-peak) magnitude. Due to their inherit susceptibility to electrical noise, these signals are most commonly differential pairs, wired into the SIN+, SIN-, COS+, and COS- inputs for the channel. Differential signals can enhance immunity by providing common mode noise rejection. Single-ended inputs can also be used, wired into the SIN+ and COS+ inputs for the channel, with the SIN- and COS- inputs connected directly to the 2.5 V reference (pin #7).

A good quality shielded cable with twisted-pair shielded conduits is highly recommended for sinusoidal encoder applications.



Standard Sinusoidal Configuration

The sinusoidal encoder signals are interpolated in the ASIC (hardware); the resulting data is brought into the encoder conversion table (ECT) as a single 32-bit word without any scaling:

```
EncTable[1].type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].ServoCapt.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 0
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1

PowerBrick[0].Chan[0].AtanEna = 1
Motor[1].ServoCtrl = 1
Motor[1].EncType = 6
```

Standard Sinusoidal Counts per line

With the standard interpolator option:

- A rotary encoder with 1,024 sine/cosine periods per revolution produces:
  1,024 x 16,384 = 16,777,216 counts / revolution
- A 20 µm linear encoder produces
  16,384 / 0.020 = 819,200 counts / mm

Standard Sinusoidal Bias Correction

The Power Brick LV has the capability of correcting for biases of the cosine / sine signals. These corrections are suitable when interpolating in the Gate3 without the ACI (Auto Correcting Interpolator) option. This procedure is described in the Sinusoidal Encoder Bias Corrections section of this manual.

Standard Sinusoidal Encoder Count Error

With Sinusoidal encoders, the Power Brick LV has the capability of trapping encoder count (loss) errors. This is described in detail in the Encoder Count Error section of this manual.

Standard Sinusoidal Encoder Loss Detection

Warning

Loss of the feedback sensor signal is potentially a very dangerous condition in closed-loop control, because the servo loop no longer has any idea what the true physical position of the motor is – usually it thinks it is "stuck" – and it can react wildly, often causing a runaway condition.

With Sinusoidal encoders, the Power Brick LV has the capability of detecting the loss of an encoder signal. This is described in detail in the Encoder Loss Detection section of this manual.

Note

Note the distinction between the encoder count error, which reports loss of counts due to bad transitions of the quadrature signals, and encoder loss, which indicates that one or more quadrature / sinusoidal signals are missing.

ACI Sinusoidal Configuration

```
EncTable[1].type = 7
EncTable[1].pEnc = PowerBrick[0].Chan[0].ServoCapt.a
EncTable[1].pEnc1 = PowerBrick[0].Chan[0].AtanSumOfSqr.a
EncTable[1].index1 = 0
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].index6 = 0
EncTable[1].ScaleFactor = 1

Gate3[0].AdcEncHeaderBits = 0
Gate3[0].AdcEncStrobe = $800000
Gate3[0].Chan[0].AtanEna = 1
Motor[1].ServoCtrl = 1
Motor[1].EncType = 7
```

ACI Sinusoidal Counts per line

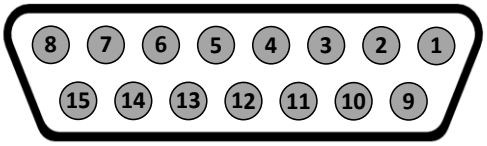
With the ACI interpolator option:

- A rotary encoder with 1,024 sine/cosine periods per revolution produces
  1,024 x 65,536 = 67,108,864 counts / revolution
- A 20 µm linear encoder produces
  65,536 / 0.020 = 3,276,800 counts / mm

Analog Resolver

If option K and/or L has a value of R, the Power Brick LV can accept resolver encoder input (up to 5 $V_{peak-peak}$) and provide interpolated position data.

														K	L				
P	B	L		-	A			0			-								0

X1-X8: D-sub DA-15F Mating: D-sub DA-15M					
Pin#	**Symbol**	**Function**	**Primary Use**	**Alternate Use**	
1	SIN +	Input	Sine +		
2	COS +	Input	Cosine +		
3	CHC +	Input	Index C +		
4	**ENCPWR**	**Output**	**Encoder Power 5 VDC (max 250 mA per channel)**		
5	CHU / DIR +	In / Out	Halls U	Direction Out +	Serial Data – / AltSin +
6	CHW / PUL +	In / Out	Halls W	Step Out +	Serial Clock – / AltCos +
7	2.5V	Output	2.5 VDC Reference power		
8	PTC	Input	Motor Thermal Input		
9	SIN –	Input	Sine –		
10	COS –	Input	Cosine –		
11	CHC –	Input	Index C –		
12	**GND**	**Common**	**Common ground**		
13	CHV / DIR –	In / Out	Halls V	Direction Out –	Serial Clock + / AltSin –
14	CHT / PUL –	In / Out	Halls T	Step Out –	Serial Data + / AltCos –
15	RES EXC.	Out	Resolver Excitation Output		

Caution

The +5 VDC encoder power is limited to ~250 mA per channel. For encoders requiring more current, the +5 VDC power can be alternately brought in externally through the +5 VDC ENC connector.

Caution

Encoders requiring a voltage level other than +5 VDC (higher or lower) should be powered up using an external power supply directly into the encoder.

Setting up Resolvers

Configuring a resolver requires setting up the excitation signal control. The excitation signal control element, **PowerBrick[].ResolverCtrl**, is a 4-channel saved component:

	Excitation Signal Control
Channels 1 – 4	PowerBrick[0].ResolverCtrl
Channels 5 – 8	PowerBrick[1].ResolverCtrl

The excitation signal control element is a 32-bit element wherein the upper 12 bits carry meaningful information is broken down as follows:

	Phase Shift (Delay)								Mag.		Freq.		Reserved																			
Bit #:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Binary:	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Hex ($):	8				0				C				0				0				0				0				0			

Bits [31 – 24] specify the phase shift or delay of the excitation sine wave with respect to the phase clock. The unit of this field is 1 / 512 of an excitation cycle. This component is usually set experimentally to maximize the magnitude of the feedback signal.

Bits [23 – 22] specify the magnitude of the excitation output. The highest magnitude that does not cause saturation of the feedback ADCs (which occurs when values in the lower 16 bits of **PowerBrick[].Chan[].AtanSumOfSqr** exceed 32767) should be used.

Peak-Peak [Volts]	Value	Binary
3.2	0	00
6.2	1	01
8.8	2	10
12.2 (Max.)	3	11

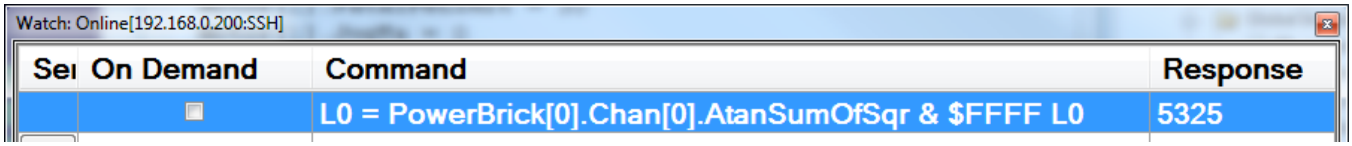
Bits [21 – 20] specify the frequency of the excitation output. The frequency that comes closest, but slightly higher, to that recommended by the resolver manufacturer should be used.

Excitation Frequency	Value	Binary
Phase Clock / 1	0	00
Phase Clock / 2	1	01
Phase Clock / 4	2	10
Phase Clock / 6	3	11

Utilizing the following expression, for channels 1 – 4 as an example:

```
GLOBAL ResExcitDelay
GLOBAL ResExcitMag
GLOBAL ResExcitFreqDiv

ResExcitMag = 3         // [0 - 3]
ResExcitFreqDiv = 0     // [0 - 3]
ResExcitDelay = 65      // [0 - 255]
PowerBrick[0].ResolverCtrl = ResExcitDelay*EXP2(24) + ResExcitMag*EXP2(22) + ResExcitFreqDiv*EXP2(20)
```

And monitoring the magnitude of the signals in the lower 16 bits of **PowerBrick[].Chan[].AtanSumOfSqr** (e.g. in the watch window):



- First, set up the excitation output magnitude, **ResExcitMag**. Start with highest (value of 3). We want the value of **AtanSumOfSqr** (lower 16 bits) to be the greatest possible.
- Set up the excitation frequency divider, **ResExcitFreqDiv**. Resolver manufacturers generally specify a minimum operating frequency. Set this typically to a value of 0, same as the phase clock.
- Set up the excitation delay (from the phase clock), **ResExcitDelay**. This value is also configured experimentally to produce the greatest possible value of the signal's magnitude, which is in the lower 16 bits of **AtanSumOfSqr**.

Configuring Resolver ECT

Once, the resolver excitation signal is set up, the encoder conversion table can be configured as follows (e.g. channel 1, motor #1):

```
EncTable[1].Type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].AtanSumofSqr.a
EncTable[1].pEnc1 = Sys.pushm
EncTable[1].index1 = 0
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].index6 = 0
EncTable[1].ScaleFactor = 1
```

The settings below are sufficient to view motor position in the position window, in counts.

```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

Resolver Counts per Revolution

With resolvers, the feedback resolution is set by the ASIC interface hardware, and produces 65,536 counts per revolution.

Resolver Absolute Power-On Position

With resolvers, the absolute position is computed directly from the upper 16 bits of the **AtanSumOfSqr** register. It is set up using the following key structure elements:

- **Motor[].pAbsPos** = **PowerBrick[0].Chan[2].AtanSumOfSqr.a**
- **Motor[].AbsPosSf** = **Motor[].PosSf**
- **Motor[].AbsPosFormat = $00001010** (Upper 16 bits)
- **Motor[].HomeOffset = (**user desired home offset value)

Note

With resolvers, it is not recommended to use PowerOnMode (value of 2) for power-on absolute position read. Instead, it is recommended to issue a **HOMEZ** command from an initialization PLC.

Bias Correction

The resolver sine and cosine signals may be corrected for biases similarly to sinusoidal encoders. This is described in the Sinusoidal Encoder Bias Corrections section of this manual.

Note

Automatic correction for signal magnitude mismatch and phase offset at the cost of additional processor time can be obtained through use of a type 4 encoder conversion table entry. Refer to Conversion Method Details, type 4 under the setting up the encoder conversion table section of the Power PMAC User Manual for more details.

Resolver Encoder Count Error

The Power Brick LV has the capability of trapping encoder count (loss) errors for resolvers. This is described in detail in the Encoder Count Error section of this manual.

Resolver Encoder Loss Detection

Warning

Loss of the feedback sensor signal is potentially a very dangerous condition in closed-loop control, because the servo loop no longer has any idea what the true physical position of the motor is – usually it thinks it is "stuck" – and it can react wildly, often causing a runaway condition.

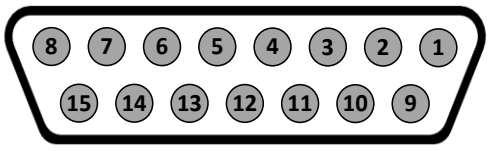
With Resolvers, the Power Brick LV has the capability of detecting the loss of an encoder signal. This is described in detail in the Encoder Loss Detection section of this manual.

Note

Note the distinction between the encoder count error, which reports loss of counts due to bad transitions of the quadrature signals, and encoder loss, which indicates that one or more quadrature / sinusoidal signals are missing.

Serial Encoders with Gate3

The Power Brick LV, in its standard configuration, accepts a variety of serial encoder protocols. These protocols are built into the DSPGate3. This section discusses the configuration of these serial encoders.

X1-X8: D-sub DA-15F Mating: D-sub DA-15M								
Pin#	**Symbol**	**Function**	**HiperFace**	**SSI EnDat**	**Panasonic**	**Mitutoyo**	**Sigma II/III/V/VII**	**Tamagawa**
1								
2								
3	ENA –	Output	–					$\overline{\text{SENA}}$
4	**ENCPWR**	**Output**	**Encoder Power 5 VDC (max 250 mA per channel)**					
5	DATA –	In / Out	DAT–	DAT–	$\overline{\text{PS}}$	MRR	SDI blu/blk	$\overline{\text{SD}}$
6	CLOCK –	Output	–	CLK–	–	–	–	–
7	2.5V	Output	2.5 VDC – Reference					
8	PTC	Input	Motor Thermal Input					
9								
10								
11	ENA +	Output	–					SENA
12	**GND**	**Common**	**Common Ground**					
13	CLOCK +	Output	–	CLK+	–	–	–	–
14	DATA +	In / Out	DAT+	DAT+	PS	MR	SDO blu	SD
15								

Caution

The +5 VDC encoder power is limited to ~250 mA per channel. For encoders requiring more current, the +5 VDC power can be alternately brought in externally through the +5 VDC ENC connector.

Caution

Encoders requiring a voltage level other than +5 VDC (higher or lower) should be powered directly from an external power supply.

Note

Quadrature / sinusoidal encoders can be wired and processed simultaneously with serial encoders on the same channel.

Pins #5, 6, 13, and 14 of the encoder feedback connectors (X1 – X8) share multiple functions: only one of these functions (per channel) can be used – configured in software – at one time:

- Hall sensor inputs (default configuration).
- Pulse and direction PFM output signals (enable using **PowerBrick[].Chan[].OutFlagD**).
- Serial encoder inputs (enable using **PowerBrick[].SerialEncEna**).
- Serial encoder inputs (enable using bit 10 of **ACC84B[].SerialEncCmd** with ACC-84B).
- Quadrature encoder inputs (serial encoder input must be disabled).
- Alternate Sinusoidal encoder inputs (with sinusoidal encoder option).

Note

Each channel is independent of the other channels and can have its own use for these pins.

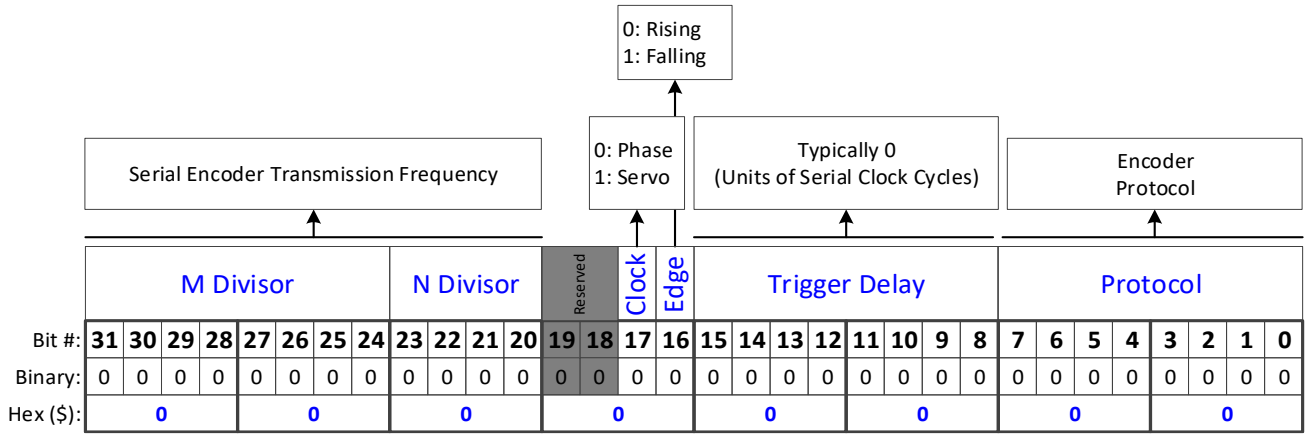
Configuring a serial encoder requires the programming of two essential structure elements, and the enabling of the serial encoder line:

- The Serial Encoder Control word, **PowerBrick[].SerialEncCtrl**
- The Serial Encoder Command word, **PowerBrick[].Chan[].SerialEncCmd**
- **PowerBrick[].Chan[].SerialEncEna = 1**

Serial Encoder Control with Gate3

The Serial Encoder Control is a 32-bit, 4-channel (1 – 4, or 5 – 8), structure element. It specifies the protocol type, delay compensation time, trigger edge, trigger clock, and transmission frequency of the 4 serial encoder channels.

	Serial Encoder Control Elements
Channels 1 – 4	PowerBrick[0].SerialEncCtrl
Channels 5 – 8	PowerBrick[1].SerialEncCtrl



Bits [31 – 20] specify the serial interface transmission frequency. This frequency (or range) is usually specified by the encoder manufacturer and programmed by the user or pre-defined by the protocol.

Bit 17 specifies the trigger source; Phase clock is recommended (value 0).

Bit 16 specifies the active edge; rising edge is recommended (value 0).

Bits [15 – 8] specify the trigger delay (in units of serial clock cycles).

Bits [3 – 0] specify the encoder protocol of the serial encoder:

Protocol	Value
–	0
–	1
SSI	2
EnDat	3

Protocol	Value
Hiperface	4
Sigma I	5
Sigma II/III/V	6
Tamagawa	7

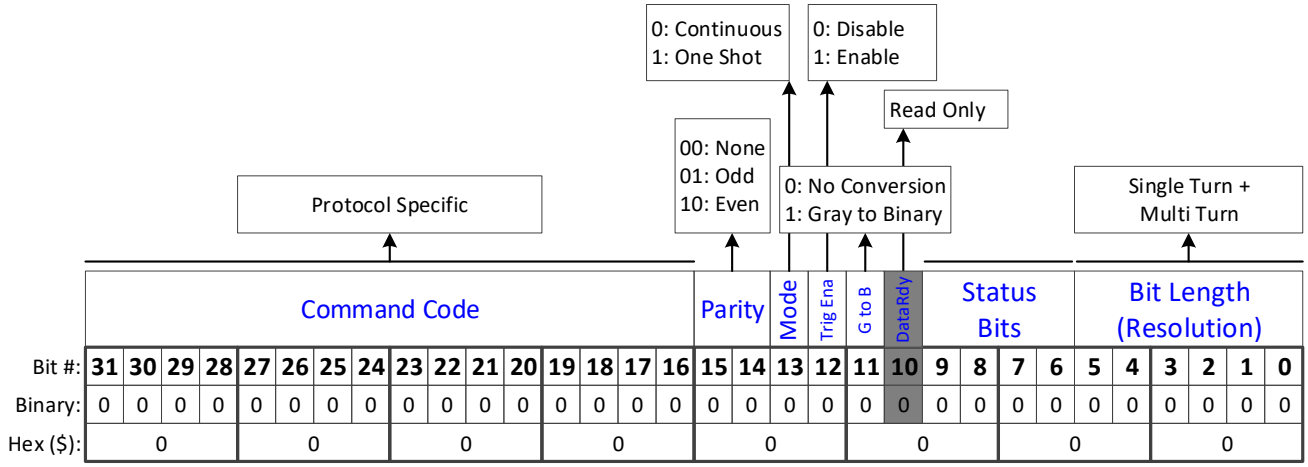
Protocol	Value
Panasonic	8
Mitutoyo	9
Kawasaki	10 ($A)
–	11 ($B)

Protocol	Value
–	12 ($C)
–	13 ($D)
–	14 ($E)
SW Ctrl	15 ($F)

Serial Encoder Command with Gate3

The Serial Encoder Command is a 32-bit, channel specific, structure element. It specifies the bit length (resolution), status bits, data type, conversion method, trigger enable, trigger mode, parity, and command code of the serial encoder channel.

Ch.#	Serial Encoder Command	Ch. #	Serial Encoder Command
1	PowerBrick[0].Chan[0].SerialEncCmd	5	PowerBrick[1].Chan[0].SerialEncCmd
2	PowerBrick[0].Chan[1].SerialEncCmd	6	PowerBrick[1].Chan[1].SerialEncCmd
3	PowerBrick[0].Chan[2].SerialEncCmd	7	PowerBrick[1].Chan[2].SerialEncCmd
4	PowerBrick[0].Chan[3].SerialEncCmd	8	PowerBrick[1].Chan[3].SerialEncCmd



Bits [31 – 16] specify the command code. This field is protocol specific.

Bits [15 – 14] specify the parity. This field is protocol specific.

Bit 13 specifies the trigger mode.

Bit 12 is the trigger enable toggle.

Bit 11 enables Gray code to binary conversion. This field is protocol specific.

Bit 10 is the data ready bit, read only.

Bits [9 – 6] specify the encoder status field. This field is protocol specific.

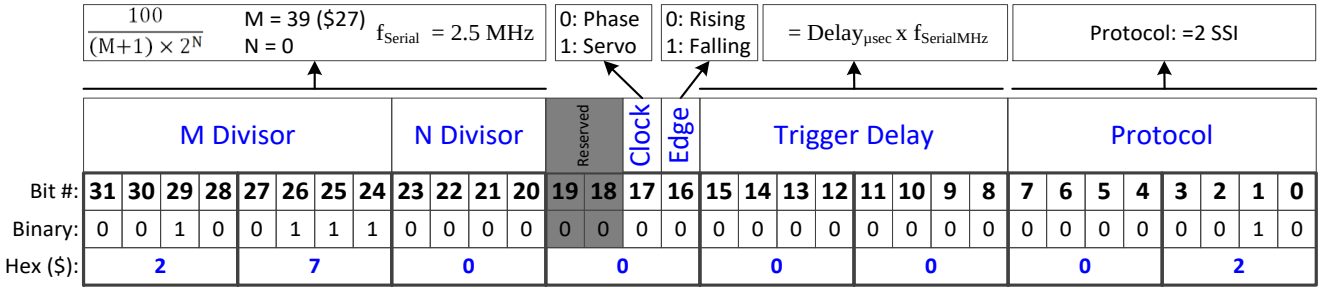
Bits [5 – 0] specify the serial encoder bit length (single-turn + multi-turn).

Following, are examples for setting up the control and command words for each of the supported protocols. Also, the resulting data registers and their format.

SSI Configuration Example with Gate3

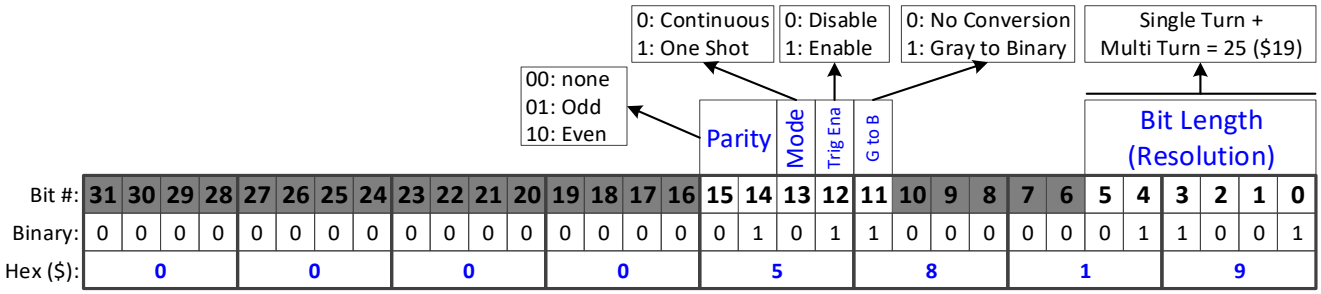
➢ **SERIAL ENCODER CONTROL – SSI**

No trigger delay, rising edge of phase, and 2.5 MHz transmission



➢ **SERIAL ENCODER COMMAND – SSI**

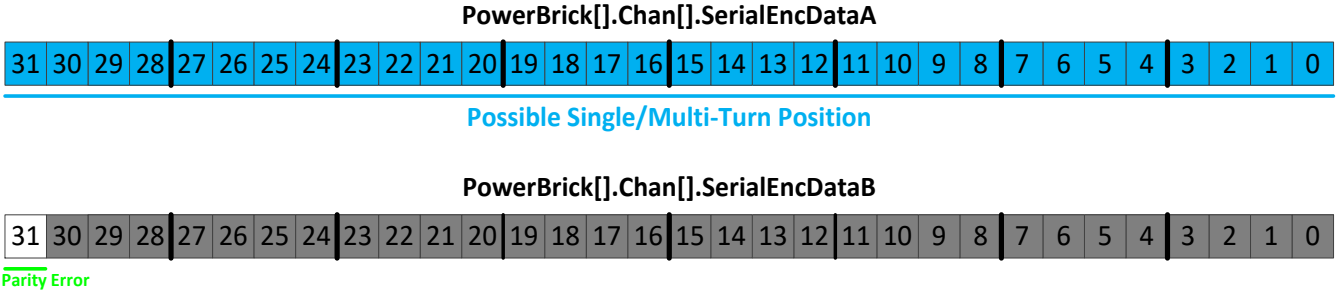
A 25-bit SSI encoder in Gray code, with odd parity




```
PowerBrick[0].SerialEncCtrl = $27000002
PowerBrick[0].Chan[0].SerialEncCmd = $5819
PowerBrick[0].Chan[0].SerialEncEna = 1
```

➢ **SERIAL DATA REGISTERS – SSI**

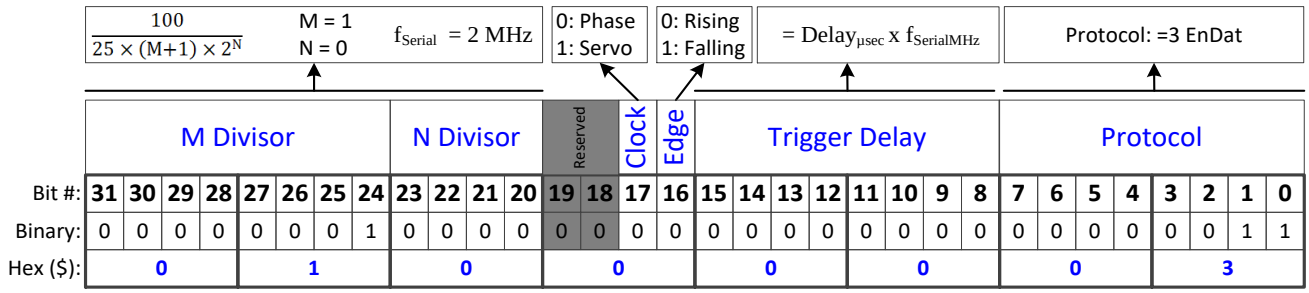
The resulting position data, status, and error bits for SSI are found in the following Serial Data Registers:



EnDat 2.1/2.2 Configuration Example with Gate3

➢ **SERIAL ENCODER CONTROL – ENDAT 2.1/2.2**

No trigger delay, rising edge of phase, and 2.0 MHz transmission



➢ **SERIAL ENCODER COMMAND – ENDAT 2.1/2.2**

The DSPGate3 interface to EnDat supports four 6-bit command codes:

- 000111 ($7) for reporting position (EnDat2.1/2.2).
- 101010 ($2A) for resetting the encoder (EnDat2.1/2.2).
- 111000 ($38) for reporting position with possible additional information (EnDat 2.2 only)
- 101101 ($2D) for resetting the encoder (EnDat 2.2 only)

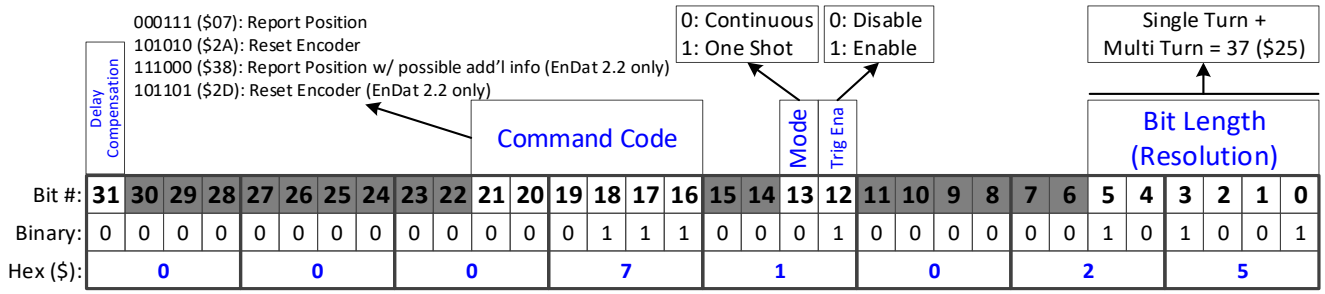
Note

By the EnDat standard, EnDat 2.2 encoders should be able to accept and process EnDat 2.1 command codes. However, not all encoders sold as meeting the EnDat 2.2 standard can do this.

Note

With the Power Brick LV, EnDat additional information is supported via the (optional) ACC-84B serial interface.

A 37-bit EnDat 2.2 encoder for continuous position reporting:




```
PowerBrick[0].SerialEncCtrl = $1000003
PowerBrick[0].Chan[0].SerialEncCmd = $71025
PowerBrick[0].Chan[0].SerialEncEna = 1
```

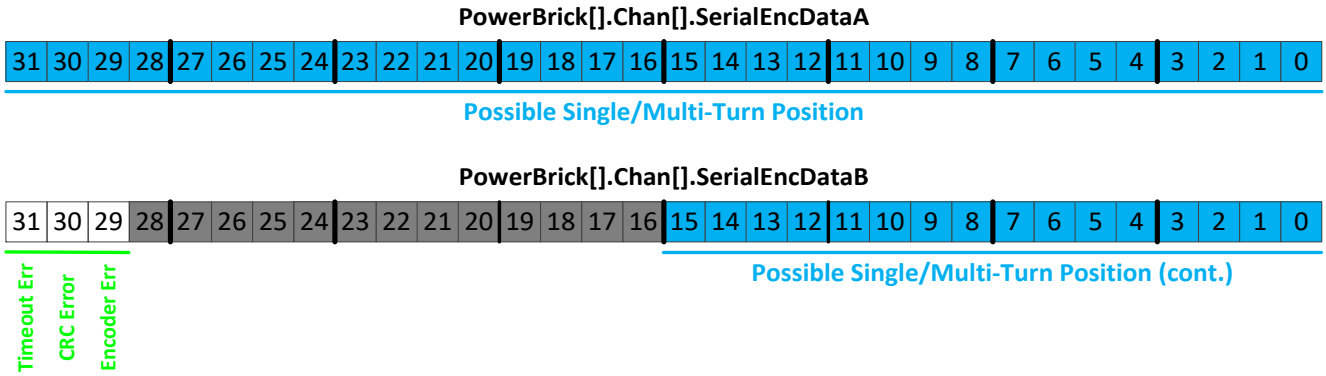
With EnDat 2.2, bit 31 is the StartDelayComp control bit. Setting this bit to 1 starts a delay identification and compensation cycle which measures the propagation delay between the encoder and the controller. The delay is measured three times and the average is used in the compensation. When these calculations are done, the StartDelayComp bit 31 is automatically cleared. This delay identification operation must be performed after every power-up cycle. Delay compensation permits high bit transmission rates over very long cables.

To perform the delay identification and compensation cycle on this encoder, set PowerBrick[].Chan[].SerialEncCmd = $80071025, then wait for bit #31 to clear.

This same encoder can be reset with a command code of $2A sent in one-shot mode, so by setting PowerBrick[].Chan[].SerialEncCmd = $2A3025.

- **SERIAL DATA REGISTERS – ENDAT 2.1/2.2**

The resulting position data, status, and error bits for EnDat are found in the following Serial Data Registers:

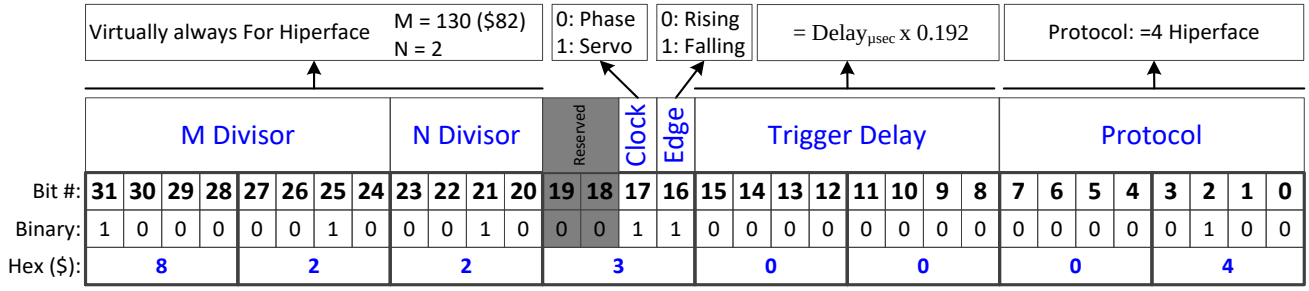


Hiperface Configuration Example with Gate3

➢ **SERIAL ENCODER CONTROL – HIPERFACE**

Because there is no explicit clock signal with Hiperface, the serial clock frequency is set 20 times higher than the bit transmission frequency to "oversample" the input data stream. For the default 9600 baud transmission of the Hiperface encoder, this clock frequency should be 9.6 x 20 = 192 kHz.

Divide the 100 MHz clock by M=130 ($83) and by 4 (N = 2) to get 192 kHz triggering on the falling edge of servo clock without delay. Since this is a "one-shot" read, the selection of the triggering clock edge does not matter much.



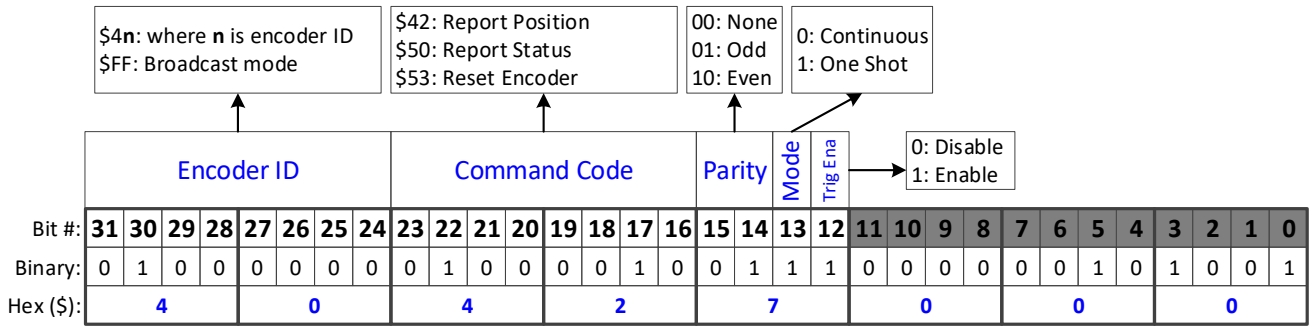
➢ **SERIAL ENCODER COMMAND – HIPERFACE**

The DSPGate3 interface to Hiperface supports three 8-bit command codes:

- $42 for reporting position.
- $50 for reporting status
- $53 for resetting the encoder

These command codes reside in the lower 8 bits of the Serial Encoder Command word. The upper 8 bits contain the address of the encoder in the interface. The Hiperface protocol permits up to 8 separate encoders to be "daisy-chained" on a single multi-drop interface. While this can be done, it is expected that each channel of the Power Brick LV will be connected to a separate individual encoder, simplifying the wiring. In this configuration, this address field can either match the encoder's address value (+ $40), or it can be set to $FF (broadcast mode).

A Hiperface encoder at user address 0 with odd parity would be set up for one-shot position reporting as follows:

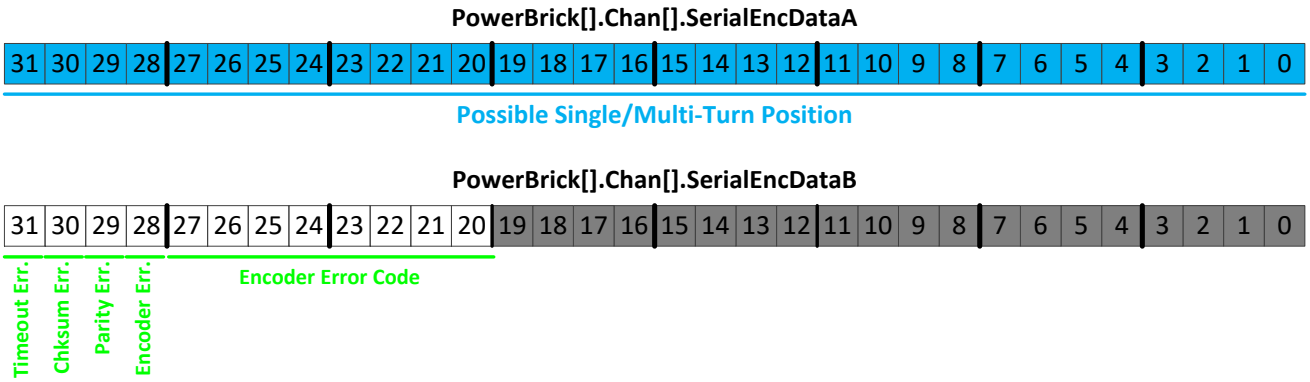


PowerBrick[].Chan[].SerialEncCmd = $40427000 or $FF427000

```
PowerBrick[0].SerialEncCtrl = $82230004
PowerBrick[0].Chan[0].SerialEncCmd = $40427000
PowerBrick[0].Chan[0].SerialEncEna = 1
```

➢ **SERIAL DATA REGISTERS – HIPERFACE**

The resulting position data, status, and error bits for Hiperface are found in the following Serial Data Registers:

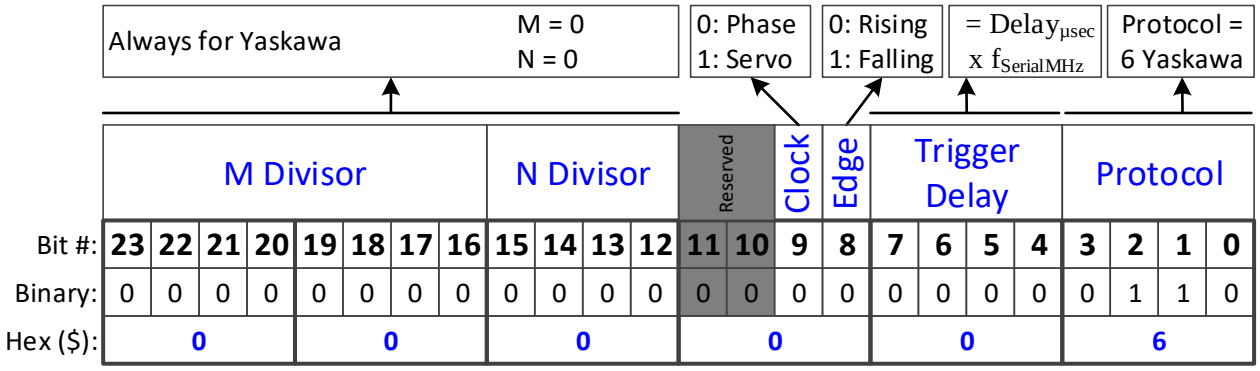


Yaskawa Sigma I Configuration Example with Gate3

➢ **SERIAL ENCODER CONTROL – SIGMA I**

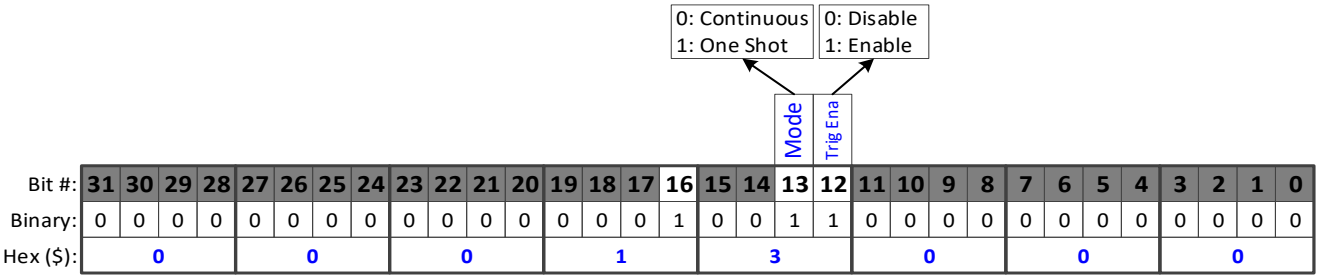
Because there is no explicit clock signal with Sigma I, the serial clock frequency is set 20 times higher than the bit transmission frequency to "oversample" the input data stream. For the default 9600 baud transmission of the Sigma I encoder, this clock frequency should be 9.6 x 20 = 192 kHz.

Divide the 100 MHz clock by M=130 ($83) and by 4 (N = 2) to get 192 kHz. triggering on the falling edge of servo clock without delay. Since this is a "one-shot" read, the selection of the triggering clock edge does not matter much. Example settings:



➢ **SERIAL ENCODER COMMAND – SIGMA I**

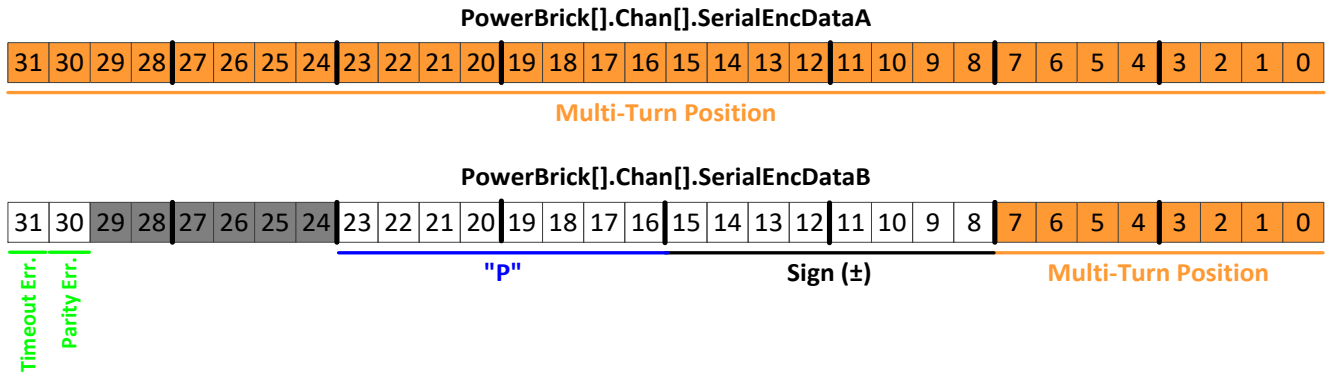
Yaskawa no longer produces Sigma I absolute encoders. However, newer generations of Yaskawa Sigma servo drives synthesize the Yaskawa Sigma I protocol for return to the controller even when using newer Sigma II, III, and V encoders. Bit 16 is set to strobe the encoder and Sigma I should use one-shot trigger, set as follows:




```
PowerBrick[0].SerialEncCtrl = $82230005
PowerBrick[0].Chan[0].SerialEncCmd = $13000
PowerBrick[0].Chan[0].SerialEncEna = 1
```

➢ **SERIAL DATA REGISTERS – SIGMA I**

The resulting position data, status, and error bits for Sigma I are found in the following Serial Data Registers:



In **SerialEncDataA**, bits [7 – 0] represent the bits of the ASCII code for the "ones digit" of the turns count, bits [15 – 8] represent bits of the "tens digit", bits [23 – 16] represent bits of the "hundreds digit", bits [31 – 24] represent bits of the "thousands digit".

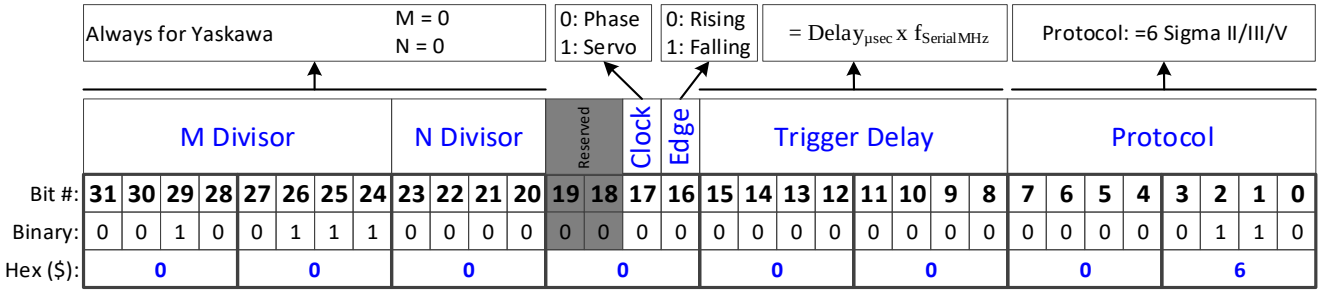
In **SerialEncDataB**, Bits [7 – 0] represent bits of the "ten-thousands digit", bits [15 – 8] represent bits of the ASCII code for the plus or minus sign, bits [23 – 16] represent bits of the ASCII code for the letter "P", bits [31 – 30] represent bits of the error field (bit 30 is a parity error; bit 31 is a timeout error).

For each of the five numeric ASCII digits, the numeric value of the digit can be obtained by subtracting 48 ($30) from the value of the ASCII code.

Yaskawa Sigma II/III/V Configuration Example with Gate3

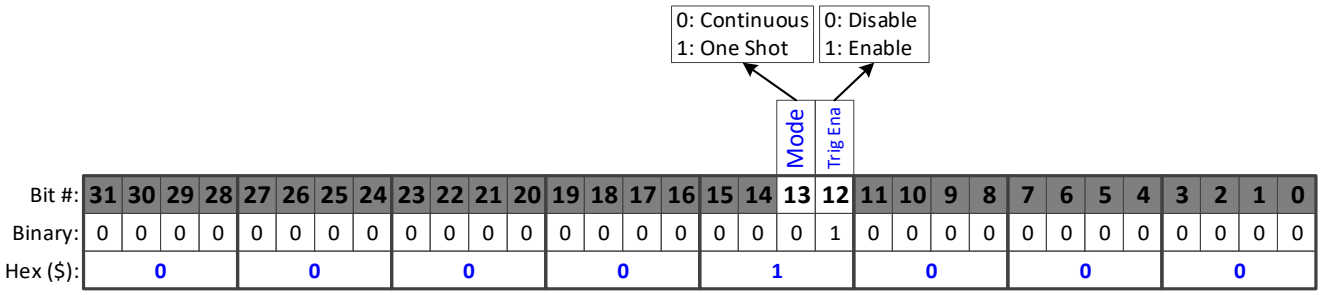
➢ **SERIAL ENCODER CONTROL – SIGMA II/III/V**

No trigger delay, rising edge of phase, and 4.0 MHz transmission:



➢ **SERIAL ENCODER COMMAND – SIGMA II/III/V**

For continuous position reporting:

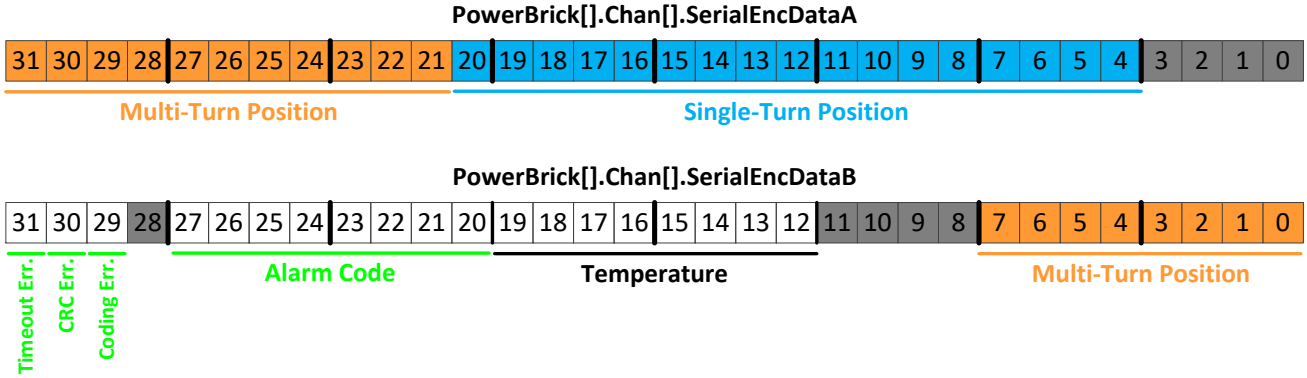



```
PowerBrick[0].SerialEncCtrl = $6
PowerBrick[0].Chan[0].SerialEncCmd = $1000
PowerBrick[0].Chan[0].SerialEncEna = 1
```

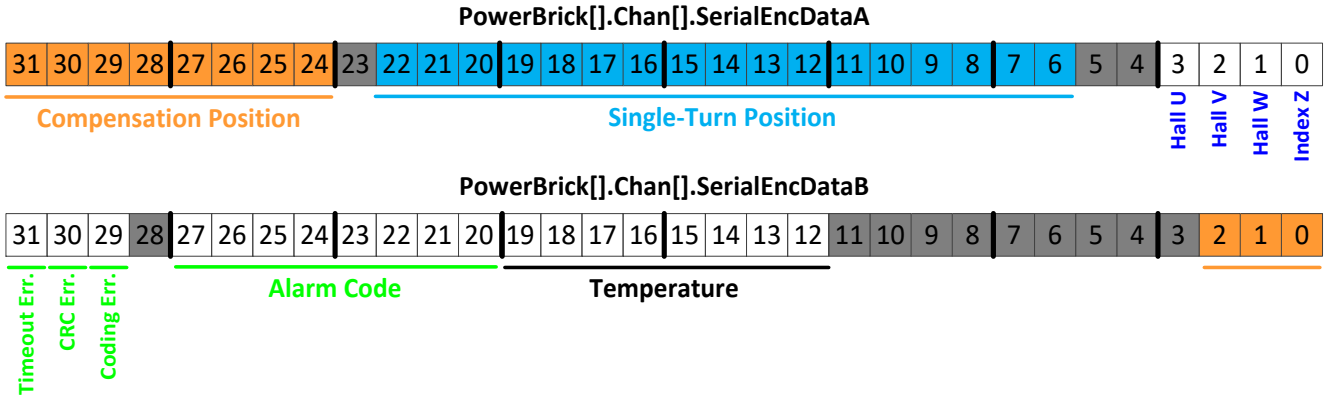
➢ **SERIAL DATA REGISTERS – SIGMA II/III/V**

The resulting position data, status, and error bits for Sigma II/III/V are found in the following Serial Data Registers:

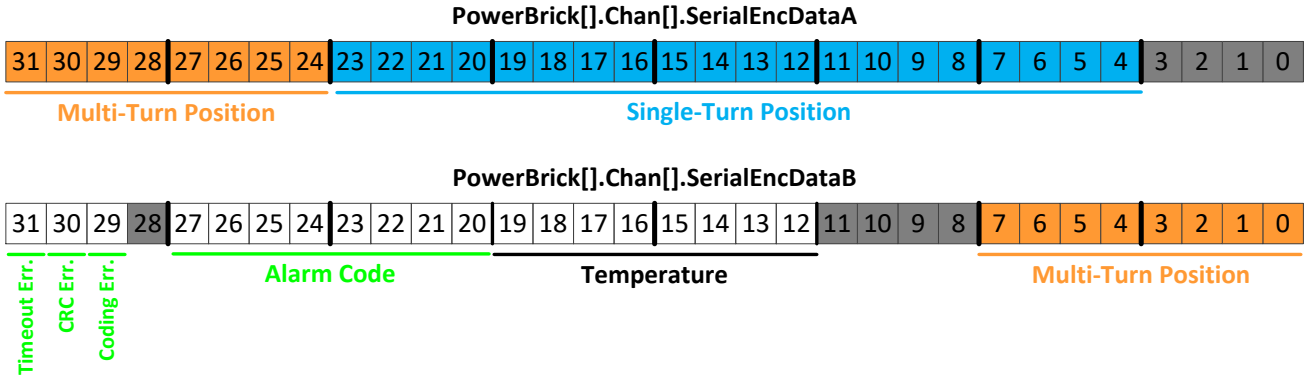
Yaskawa Sigma II (absolute 17-bit)



Yaskawa Sigma II (incremental 17-bit)



Yaskawa Sigma III/V (absolute 20-bit)



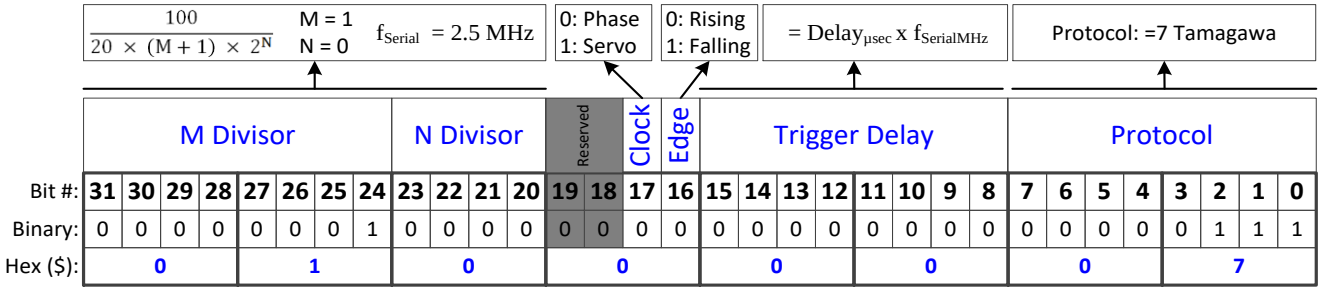
Yaskawa Sigma II/II/V Encoders Alarm Code (**SerialEncDataB**)

Bit #	Alarm Code
21	Power-on error self-detected
23	Revolution count (index to index) incorrect
26	Position reference (index) not found yet

Tamagawa FA-Coder Configuration Example with Gate3

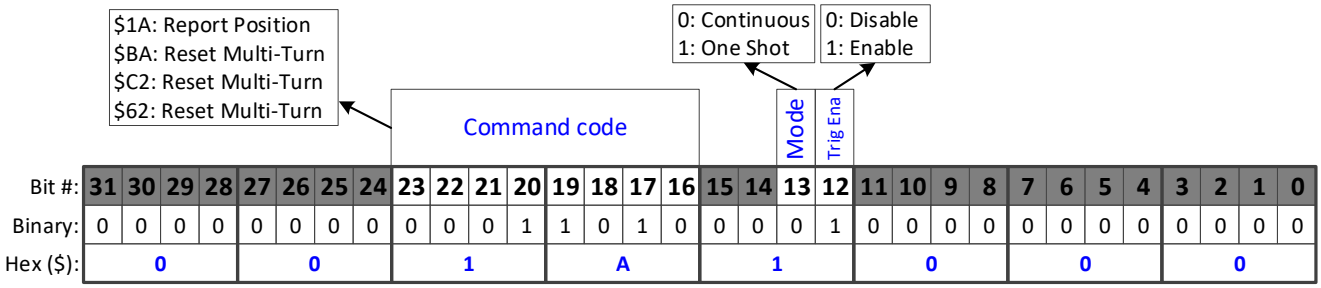
➢ SERIAL ENCODER CONTROL – TAMAGAWA FA-CODER

No trigger delay, rising edge of phase, and 2.5 MHz transmission:



➢ SERIAL ENCODER COMMAND – TAMAGAWA FA-CODER

For continuous position reporting:

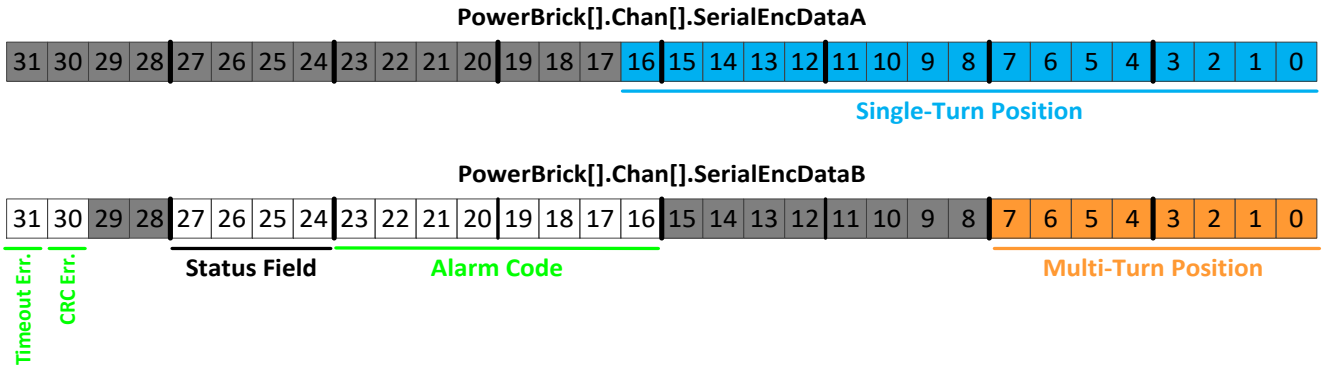


If the command code is set to $BA, $C2, or $62, the multi-turn position value in the encoder is reset to 0. This should be done in "one-shot" mode, making the element equal to $00BA3000, $00C23000, or $00623000, respectively. When the reset operation is done, the component should report as $00BA2000, $00C22000, or $00622000, respectively.

```
PowerBrick[0].SerialEncCtrl = $1000007
PowerBrick[0].Chan[0].SerialEncCmd = $1A1000
PowerBrick[0].Chan[0].SerialEncEna = 1
```

➢ SERIAL DATA REGISTERS – TAMAGAWA FA-CODER

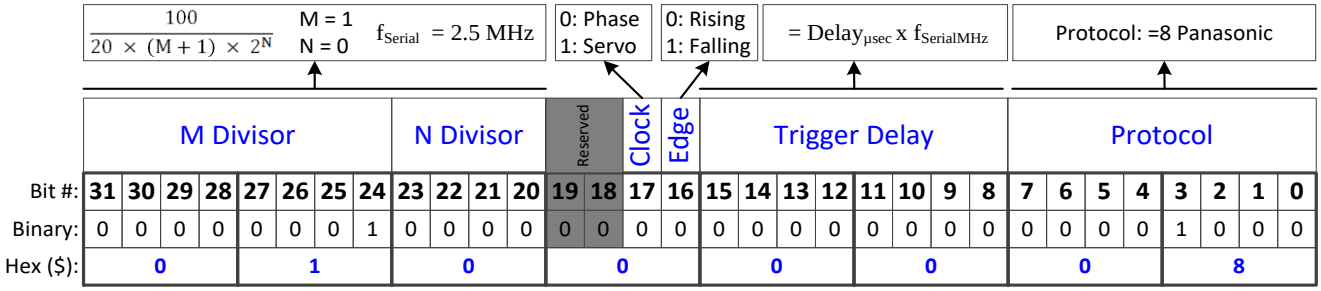
The resulting position data, status, and error bits for Tamagawa FA-Coder are found in the following Serial Data Registers:



Panasonic Configuration Example with Gate3

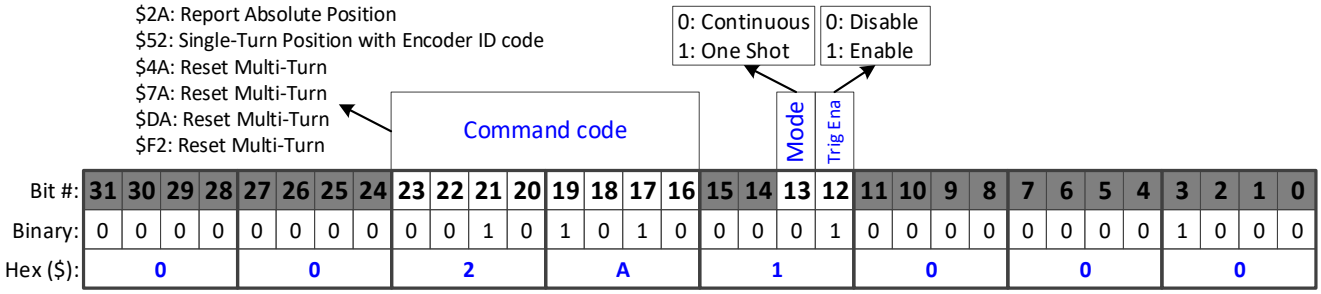
➢ **SERIAL ENCODER CONTROL – PANASONIC**

No trigger delay, rising edge of phase, and 2.5 MHz transmission



➢ **SERIAL ENCODER COMMAND – PANASONIC**

For continuous position reporting



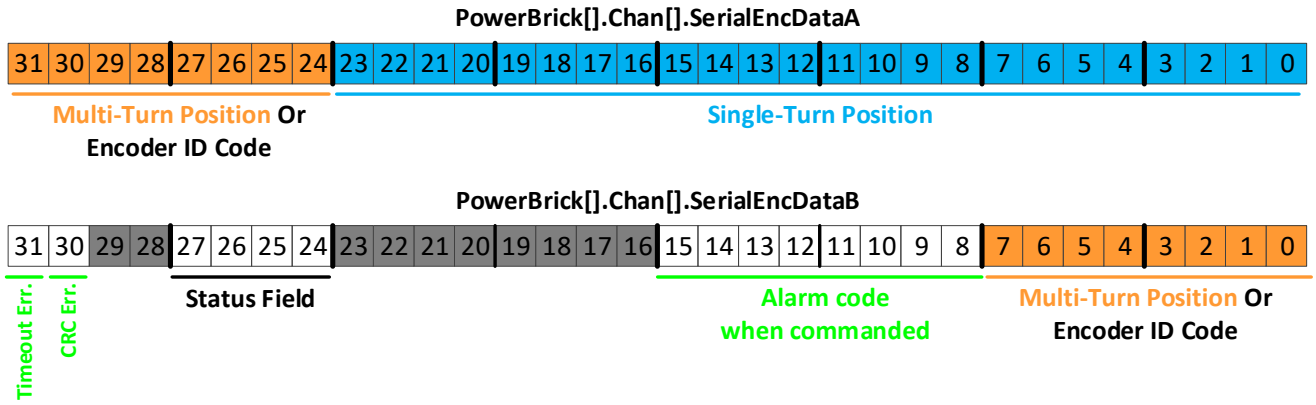
If the command code is set to $52 for single-turn position reporting with alarm code, the encoder ID value is reported where multi-turn position is normally reported.

If the command code is set to $4A, $7A, $DA, or $F2, the multi-turn position value in the encoder is reset to 0. This should be done in "one-shot" mode, making the element equal to $004A3000, $007A3000, $00DA3000, or $00F23000, respectively. When the reset operation is done, the component should report as $004A2000, $007A2000, $00DA2000, or $00F22000, respectively.

```
PowerBrick[0].SerialEncCtrl = $1000008
PowerBrick[0].Chan[0].SerialEncCmd = $2A1000
PowerBrick[0].Chan[0].SerialEncEna = 1
```

➢ **SERIAL DATA REGISTERS – PANASONIC**

The resulting position data, status, and error bits for Panasonic are found in the following Serial Data Registers:



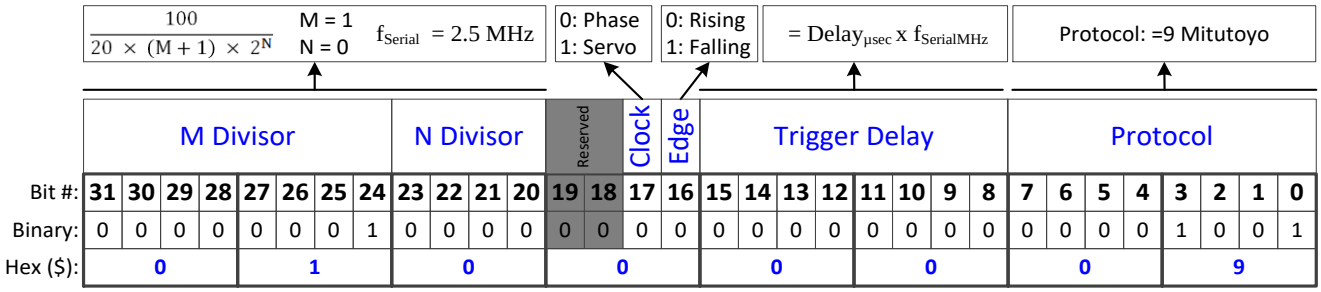
Bits 24 – 31 (**SerialEncDataA**) of the encoder ID code are fixed at a value of $11.

Bit #	Alarm Code
8	Overspeed error
9	Full resolution status; = 1 when over 100 rpm and reporting reduced resolution
10	Count Error
11	Counter Overflow
13	Multi-revolution error
14	System undervoltage error (< 2.5 V)
15	Battery low (< 3.1 V)

Mitutoyo Configuration Example with Gate3

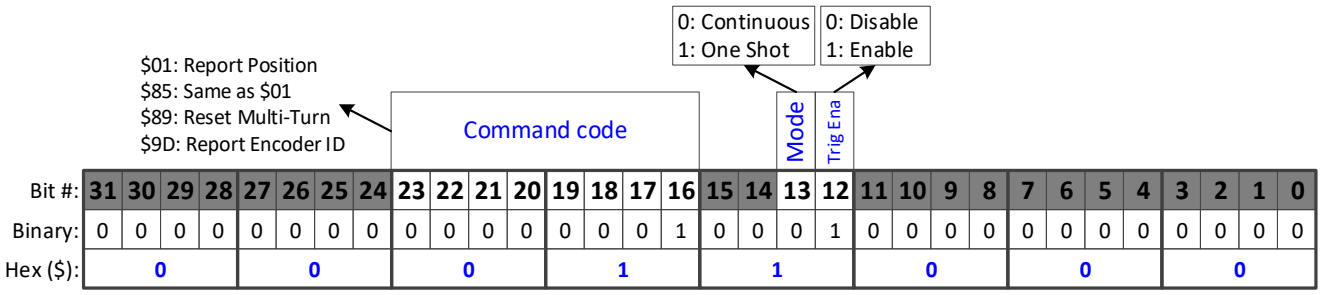
➢ **SERIAL ENCODER CONTROL – MITUTOYO**

No trigger delay, rising edge of phase, and 2.5 MHz transmission:



➢ **SERIAL ENCODER COMMAND – MITUTOYO**

For continuous position reporting:

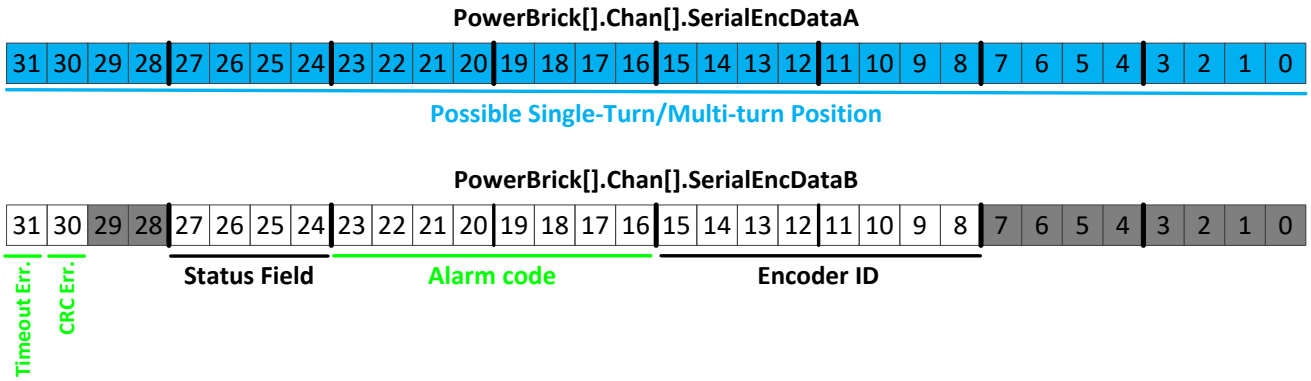


If the command code is set to $89, the multi-turn position value in the encoder is reset to 0 (after 8 cycles). If the command code is set to $9D, the encoder ID value is reported in bits 8 – 15 of **SerialEncDataB**. If the command code is set to $85, absolute position is reported, similarly to $01.

```
PowerBrick[0].SerialEncCtrl = $1000009
PowerBrick[0].Chan[0].SerialEncCmd = $11000
PowerBrick[0].Chan[0].SerialEncEna = 1
```

➢ **SERIAL DATA REGISTERS – MITUTOYO**

The resulting position data, status, and error bits for Mitutoyo are found in the following Serial Data Registers:



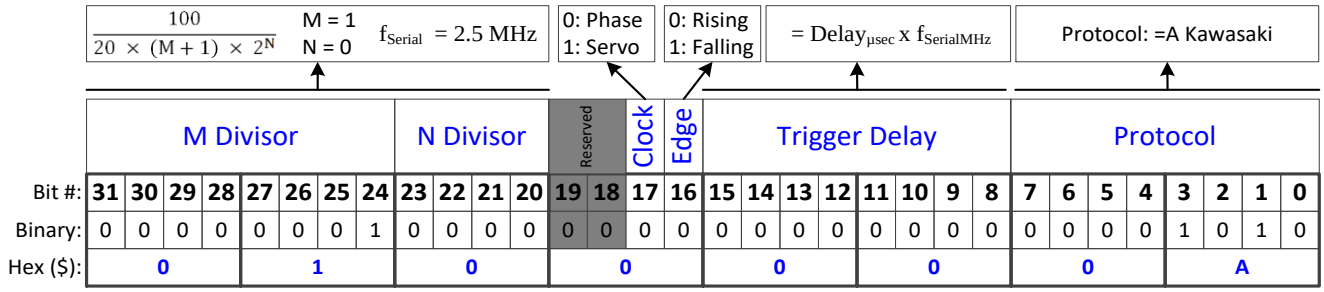
Bit #	Alarm Code
16	Initialization error
17	Mismatch of optical and capacitive sensors
18	Optical sensor error
19	Capacitive sensor error
20	CPU error (AT303); CPU/ROM/RAM error (AT503)
21	EEPROM error
22	ROM/RAM error (AT303); communication error (AT503)
23	Overspeed error

Bit #	Status Field
24	Fatal (unrecoverable) encoder error
26	Illegal command code from controller

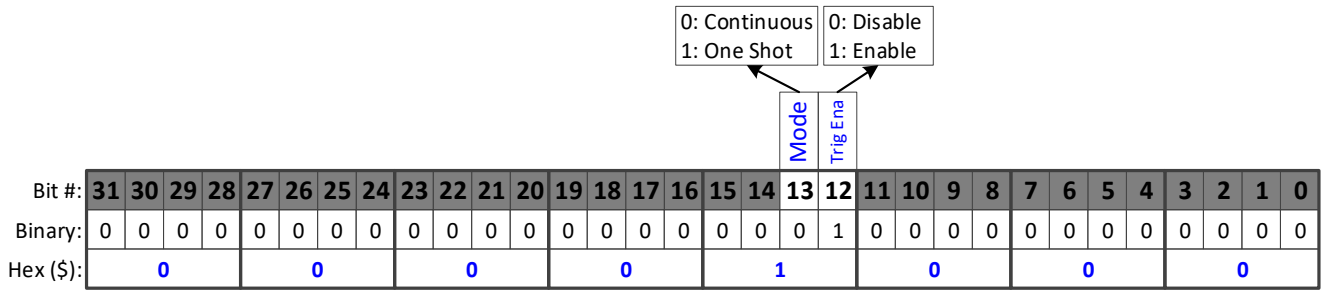
Kawasaki Configuration Example with Gate3

➢ SERIAL ENCODER CONTROL – KAWASAKI

No trigger delay, rising edge of phase, and 2.5 MHz transmission:



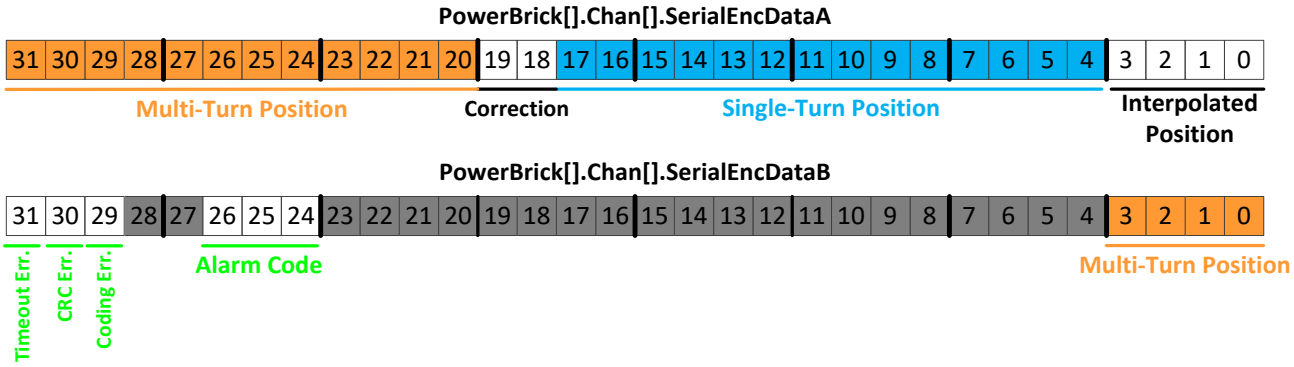
➢ SERIAL ENCODER COMMAND – KAWASAKI




```
PowerBrick[0].SerialEncCtrl = $100000A
PowerBrick[0].Chan[0].SerialEncCmd = $1000
PowerBrick[0].Chan[0].SerialEncEna = 1
```

➢ SERIAL DATA REGISTERS – KAWASAKI

The resulting position data, status, and error bits for Kawasaki are found in the following Serial Data Registers:



Bit #	Alarm code
24	Interpolator error
25	Absolute track error
26	Busy flag

Serial Encoder Ongoing Position Setup with Gate3

For the on-going "incremental" position data, it is sufficient to process whatever position data (single-turn and/or multi-turn) is available in **PowerBrick[].Chan[].SerialEncDataA**. The PMAC firmware does not require processing the entire bit length, the difference change in between servo cycles is used to compute the on-going position. This will not limit the resolution or hinder the performance. Some people may choose to use strictly the single-turn data in the Encoder Conversion Table for simplicity.

A key step is to make sure that unwanted data has been cleared and the Most Significant Bit (MSB) of the data chosen is left-shifted to bit #31 in order to handle the rollover gracefully. **EncTable[].index2** is set to the number of unwanted bits to the right of the desired data, so that a right shift can be performed to clear that unwanted data. **EncTable[].index1** is then set to the number of bits the data must be shifted left (after the right shift) to make the (MSB) of your position data bit #31.

The following settings are required to read on-going position in counts. These settings depend primarily on the location of the position data in the **SerialEncDataA**.

Structure Element	**Value**
EncTable[].type	1
EncTable[].pEnc	**PowerBrick[].Chan[].SerialEncDataA.a**
EncTable[].pEnc1	**Sys.Pushm**
EncTable[].index1	Number of bits to left shift (second operation)
EncTable[].index2	Number of bits to right shift (first operation)
EncTable[].index3	0
EncTable[].index4	0
EncTable[].index5	0
EncTable[].index6	0
EncTable[].ScaleFactor	$1 / 2^{\textbf{EncTable[].index1}}$

Structure Element	**Value**
Motor[].ServoCtrl	1
Motor[].pEnc	**EncTable[].a**
Motor[].pEnc2	**EncTable[].a**

The following are examples for setting up the Encoder Conversion Table (ECT).

Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of **SerialEncDataA**.

PowerBrick[].Chan[].SerialEncDataA

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The position data should be shifted 15 bits left (using index1) so that the Most Significant Bit (MSB) is at bit #31 to handle the rollover gracefully. Also, the scale factor should reflect the new location of the Least Significant Bit (LSB).

After Shifting

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

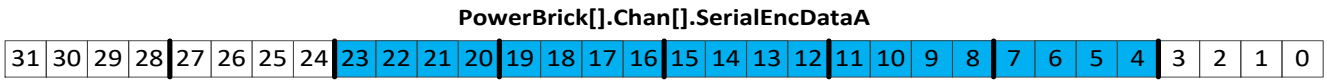
```
EncTable[1].type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 15
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1 / EXP2(15)
```

The settings below are sufficient to view motor position in the position window, in counts.

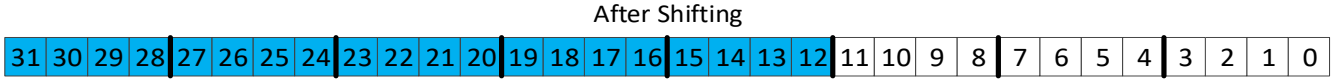
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{SingleTurn} = 2^{17} = 131{,}072$ counts per revolution for a rotary encoder. And 1/0.001 = 1,000 counts per mm for a linear encoder.

Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.



The position data should be first shifted 4 bits to the right (using **index2**) to get rid of the unwanted data. Then shifted 12 bits to the left (using **index1**), so that the (MSB) is at bit #31 to handle the rollover gracefully. Also, the scale factor should reflect the new location of the (LSB).



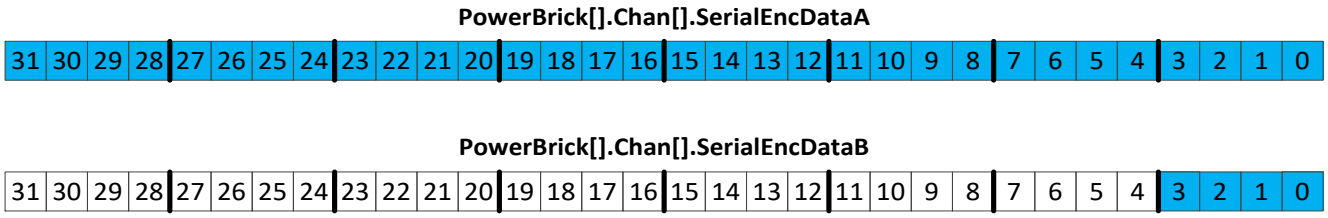

```
EncTable[1].type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 12
EncTable[1].index2 = 4
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1 / EXP2(12)
```

The settings below are sufficient to view motor position in the position window, in counts.

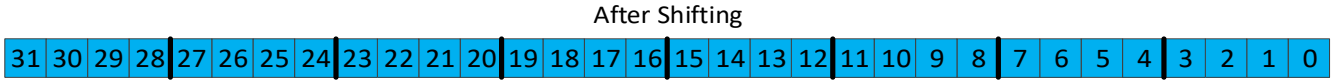
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{SingleTurn} = 2^{20} = 1{,}048{,}576$ counts per revolution for a rotary encoder. And 1 / 0.000050 = 20,000 counts per mm for a linear encoder.

Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of **SerialEncDataA** and extending to bit #3 of **SerialEncDataB**.



Reading and processing the 32 bits of position data in **SerialEncDataA** is sufficient for producing the proper ongoing position.



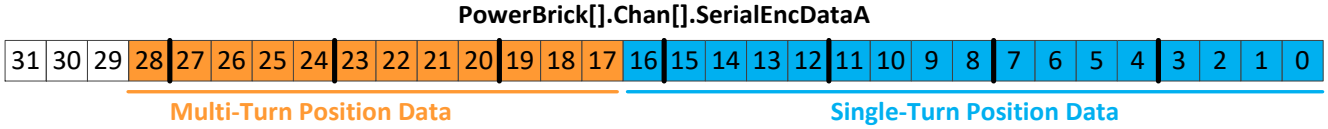

```
EncTable[1].type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 0
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1
```

The settings below are sufficient to view motor position in the position window, in counts.

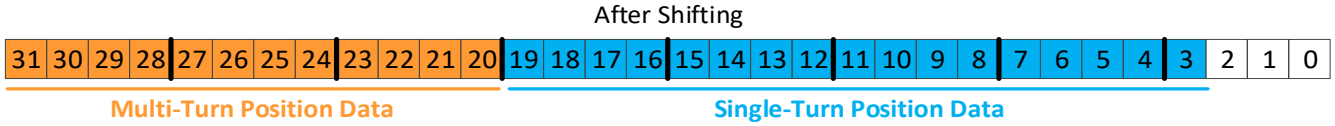
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{SingleTurn} = 2^{36} = 68,719,476,736$ counts per revolution for a rotary motor. And 1 / 0.000001 = 1,000,000 counts per mm for a linear motor.

Example 4: A 29-bit binary serial encoder with 17 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA**.



Both single-turn and multi-turn data can be used for ongoing position. The entire bit length is shifted left (using **index1**) 3 bits to place the (MSB) at bit #31.



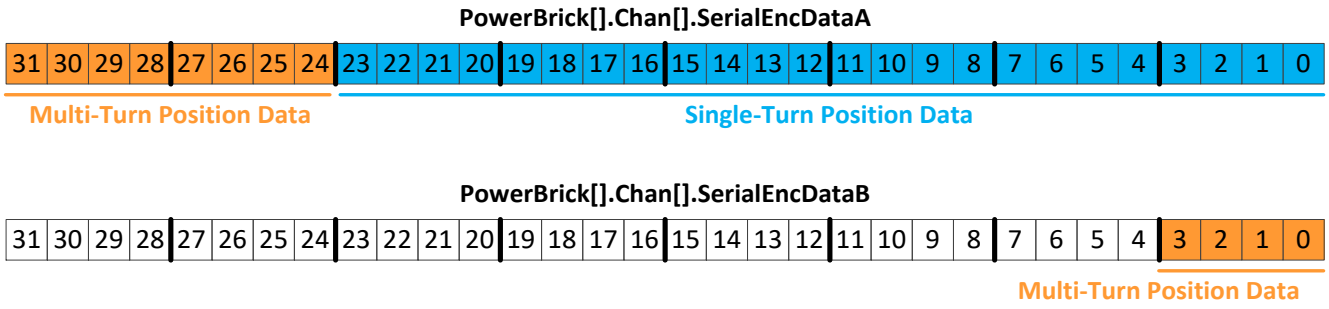

```
EncTable[1].type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 3
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1 / EXP2(3)
```

The settings below are sufficient to view motor position in the position window, in counts.

```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

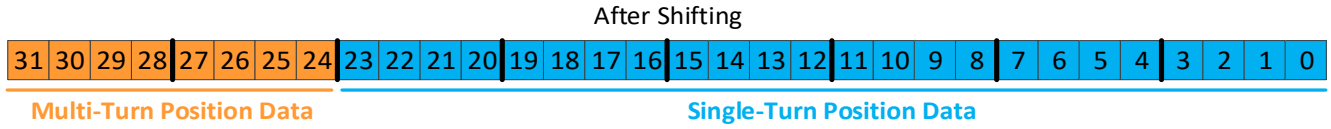
In this case, the user will see $2^{SingleTurn} = 2^{17} = 131{,}072$ counts per revolution.

Example 5: A 36-bit binary serial encoder with 24 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA** and continuously extending to bit #3 of **SerialEncDataB**.



For on-going position, we are only interested in the position data residing in **SerialEncDataA**. Some people may elect to use only the single-turn data for on-going position processing. This would require shifting to the left 8 bits (**index1 = 8**), and setting up the **EncTable[].ScaleFactor = 1 / 256**.

But, also it is possible to simply process the whole 32-bit word comprised of single-turn, and multi-turn position data with no shifting.




```
EncTable[1].type = 1
EncTable[1].pEnc = PowerBrick[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 0
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1
```

The settings below are sufficient to view motor position in the position window, in counts.

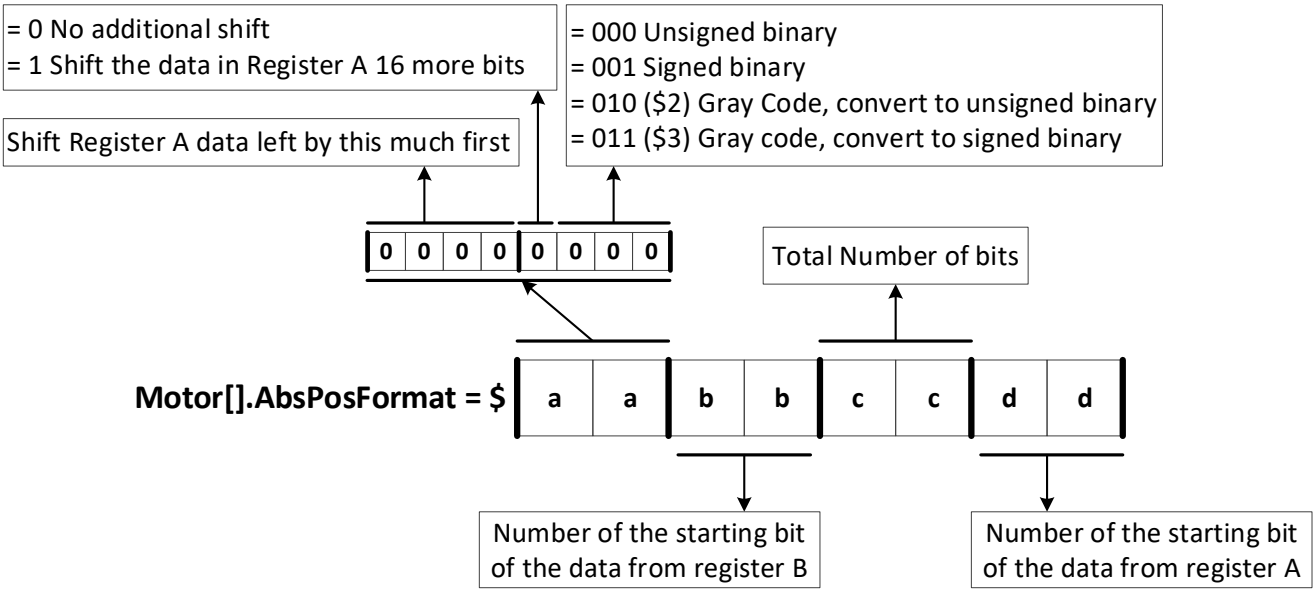
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{SingleTurn} = 2^{24} = 16{,}777{,}216$ counts per revolution.

Serial Encoder Power-on Absolute Position Setup with Gate3

The absolute position is computed directly from the serial data registers, and set up using the following key structure elements:

- **Motor[].pAbsPos**, typically = **PowerBrick[].Chan[].SerialEncDataA.a**
- **Motor[].AbsPosSf = Motor[].PosSf**
  - These settings should appear after Scaling to Engineering Units (in your motor setup file) so that **Motor[].PosSf** is already set.
- **Motor[].AbsPosFormat**:
  - Encoders with no multi-turn position data are unsigned. Rotary encoders with multi-turn position data are signed.



- **Motor[].HomeOffset = 0**

Note

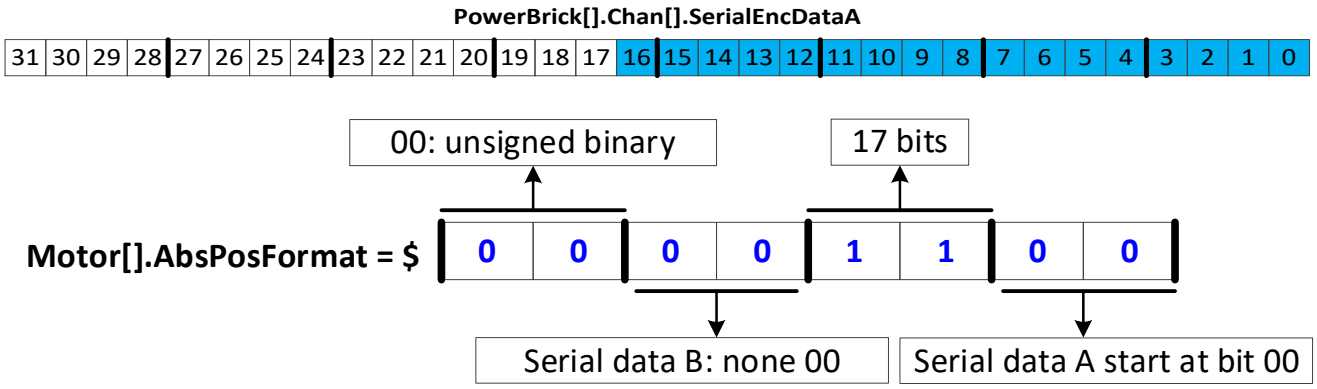
Gray code conversion should be omitted here if it had been already implemented in **PowerBrick[].Chan[].SerialEncCmd** word.

Note

Motor[].PowerOnMode bit 2 (value of 4) specifies an absolute position read on power up. Alternately, **#1HMZ** from the online terminal or a **HOMEZ 1** from a PLC can be issued to retrieve the absolute position.

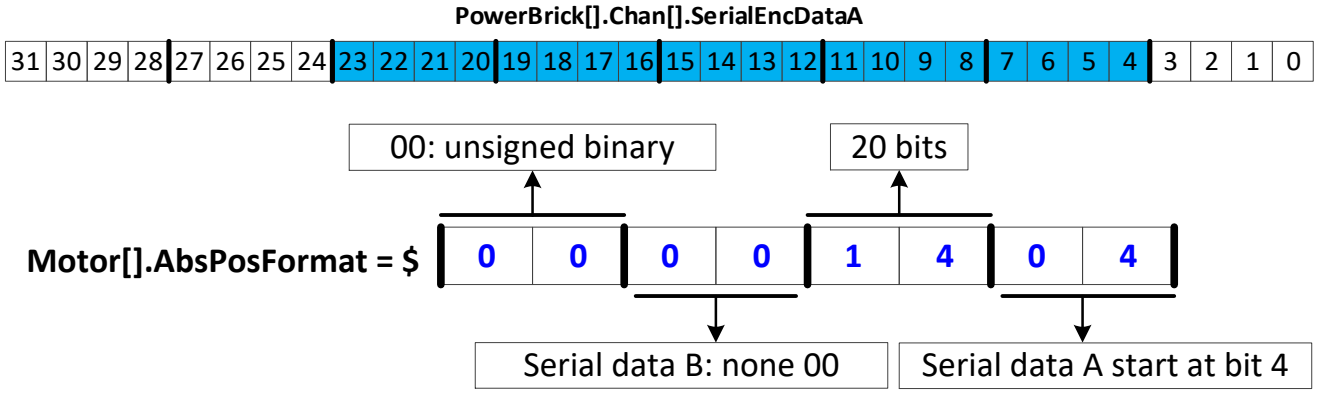
Following, are examples for setting up the absolute position read with various serial encoders. These settings depend primarily on the location of the position data in **SerialEncDataA** and **SerialEncDataB**.

Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of **SerialEncDataA**.



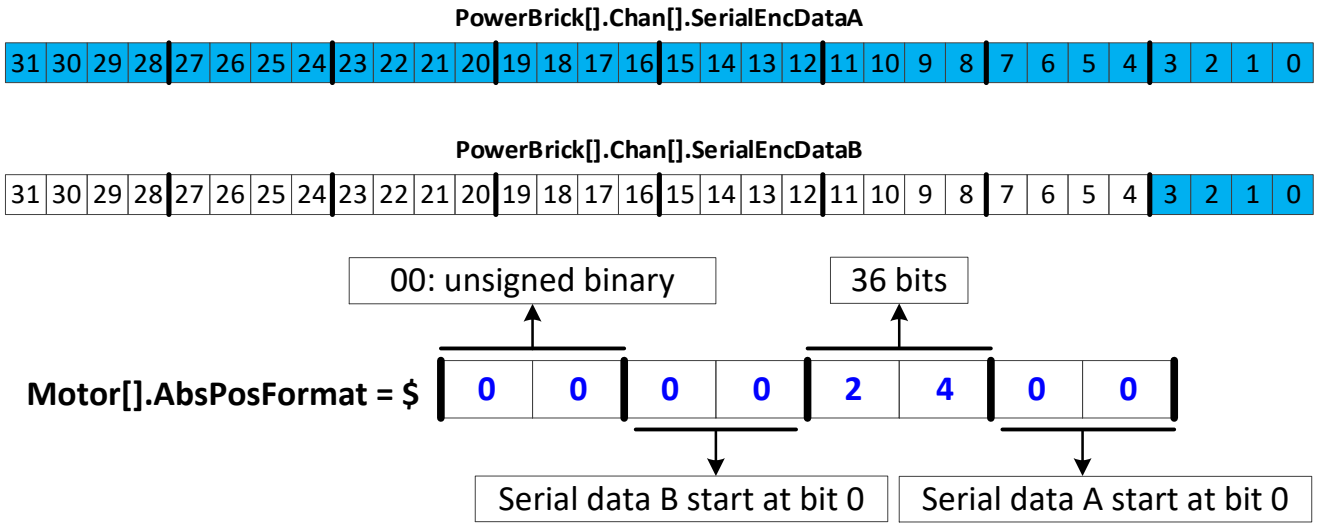

```
Motor[1].pAbsPos = PowerBrick[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $00001100
Motor[1].HomeOffset = 0
```

Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.



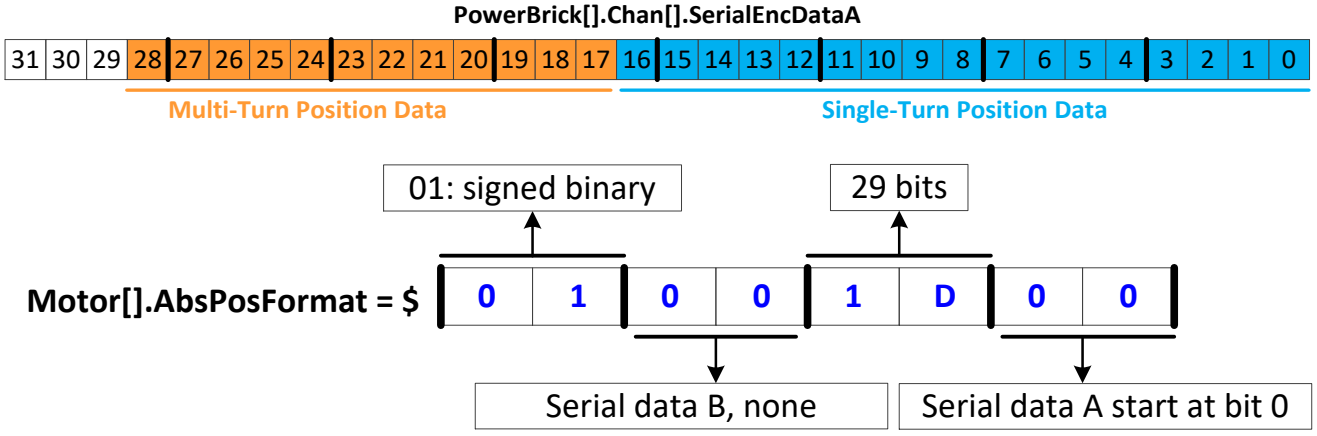

```
Motor[1].pAbsPos = PowerBrick[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $00001404
Motor[1].HomeOffset = 0
```

Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of **SerialEncDataA** and extending to bit #3 of **SerialEncDataB**.



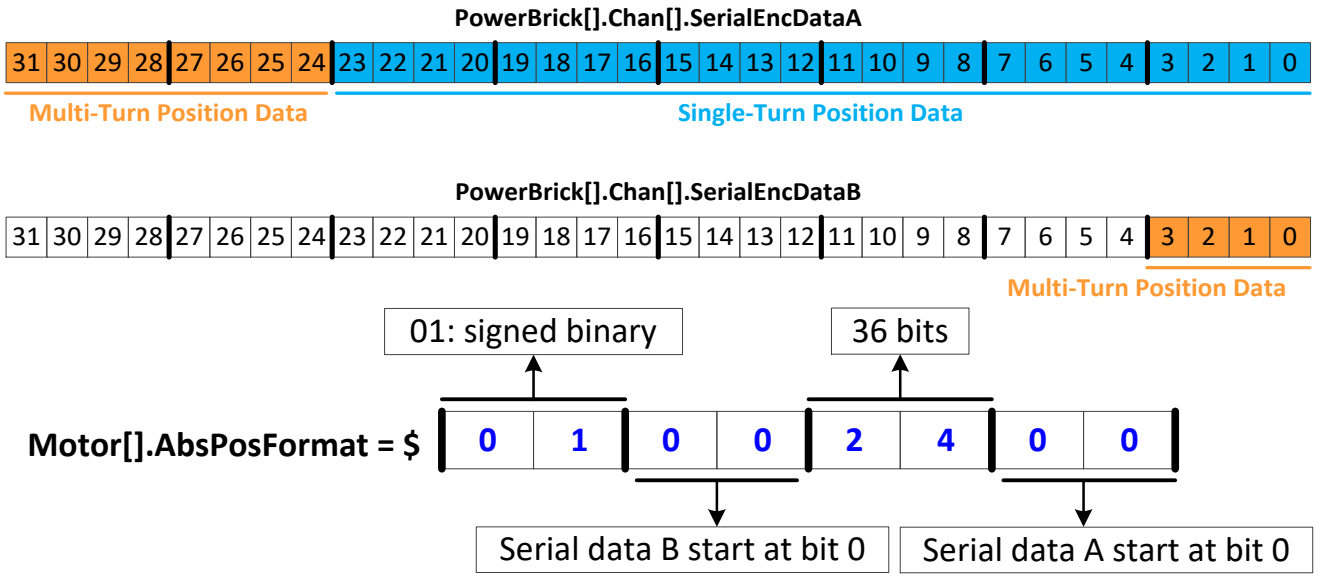

```
Motor[1].pAbsPos = PowerBrick[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $00002400
Motor[1].HomeOffset = 0
```

Example 4: A 29-bit binary serial encoder with 17 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA**.



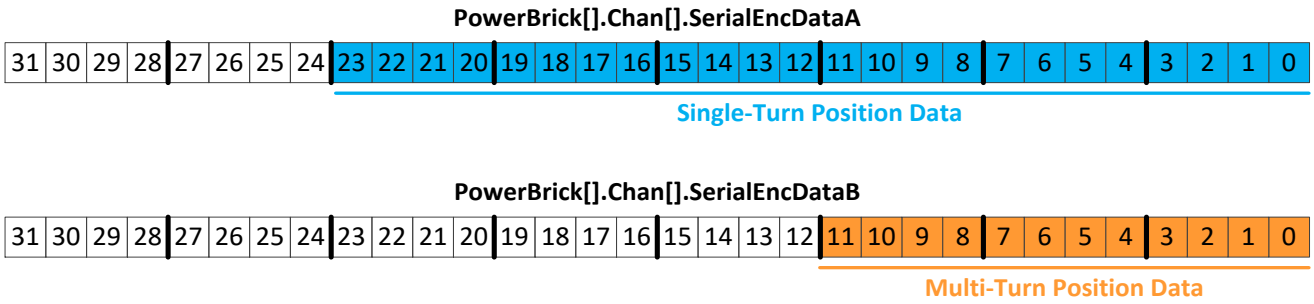

```
Motor[1].pAbsPos = PowerBrick[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $01001D00
Motor[1].HomeOffset = 0
```

Example 5: A 36-bit binary serial encoder with 24 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA** and continuously extending to bit #3 of **SerialEncDataB**.

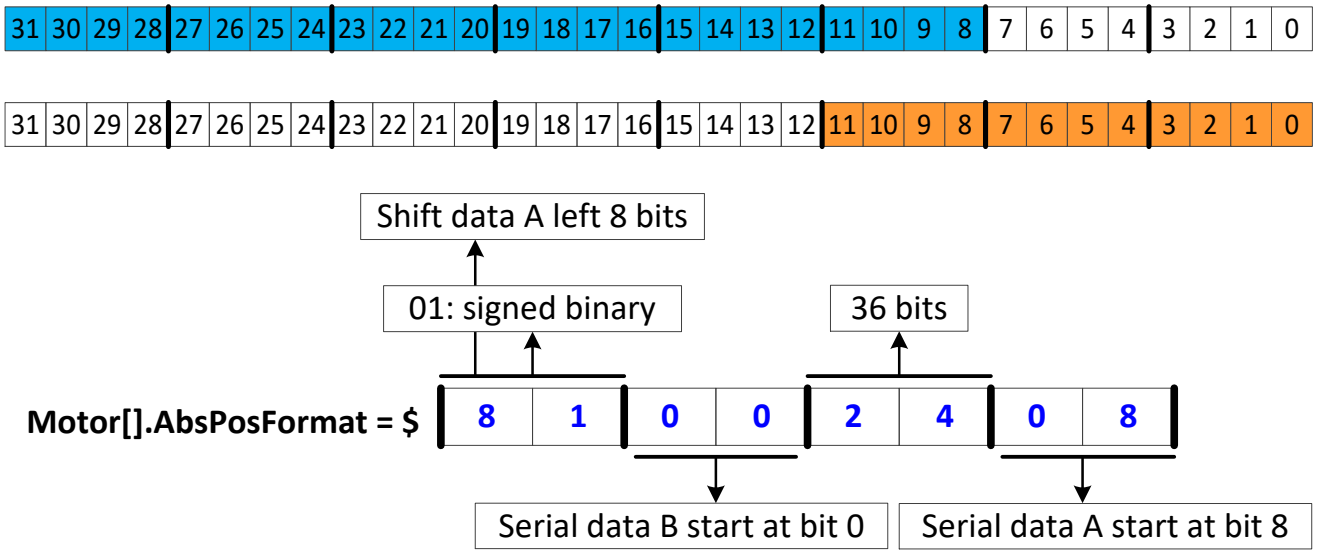



```
Motor[1].pAbsPos = PowerBrick[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $01002400
Motor[1].HomeOffset = 0
```

Example 6: A 36-bit binary serial encoder with 24 bits of single-turn data starting at bit #0 of **SerialEncDataA**, and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataB**.

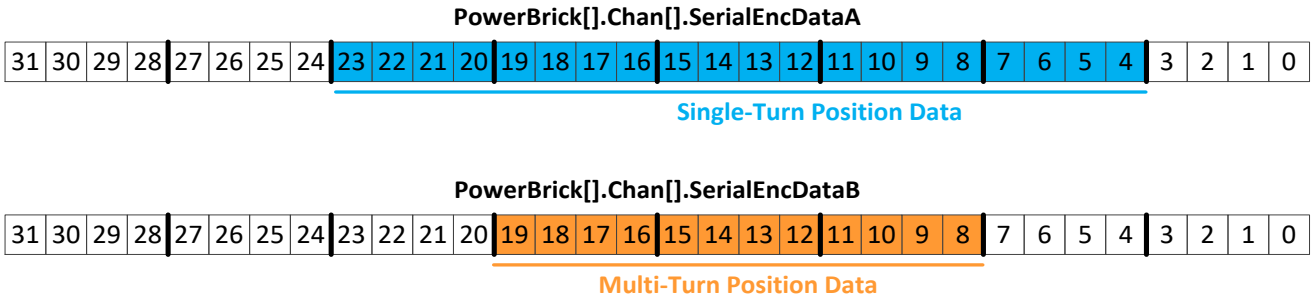


The single turn data must be shifted 8 bits left first, to make it contiguous with the multi-turn data. This shift is done using the upper 5 bits of the **$aa** byte of **Motor[].AbsPosFormat**. The data would then look like:

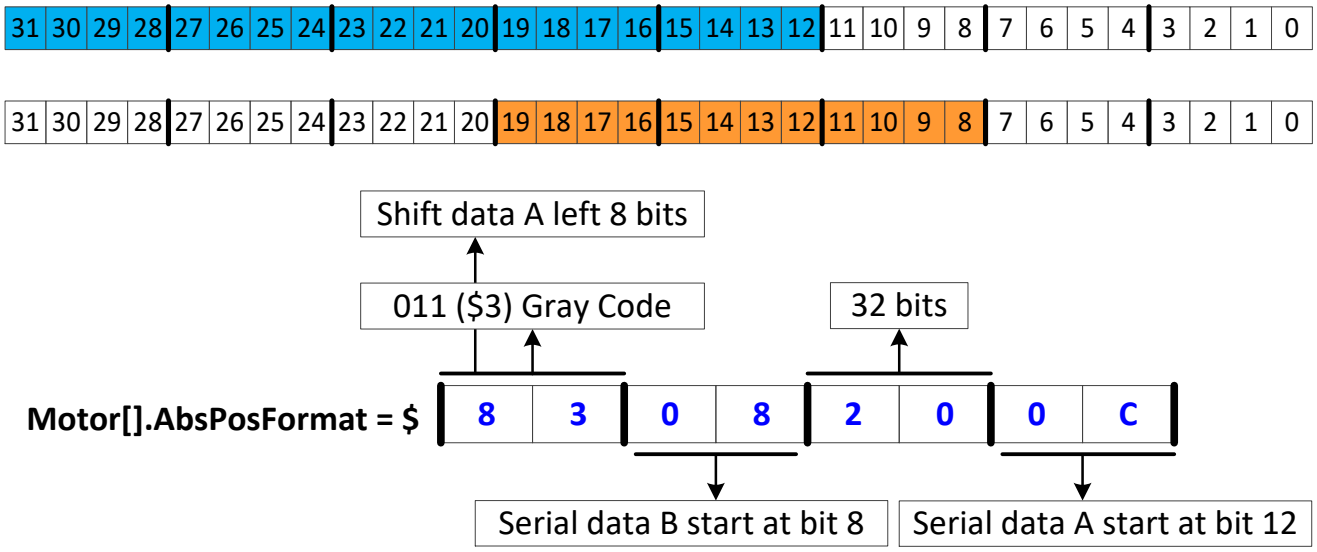



```
Motor[1].pAbsPos = PowerBrick[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $81002408
Motor[1].HomeOffset = 0
```

Example 7: A 32-bit Gray code serial encoder with 20 bits of single-turn data starting at bit #4 of **SerialEncDataA**, and 12 bits of multi-turn position data starting at bit #8 of **SerialEncDataB**.



The single turn data must be shifted 8 bits left first. This shift is done using the upper 5 bits of the **$aa** byte of **Motor[].AbsPosFormat**. The data would then look like:




```
Motor[1].pAbsPos = PowerBrick[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $8308200C
Motor[1].HomeOffset = 0
```

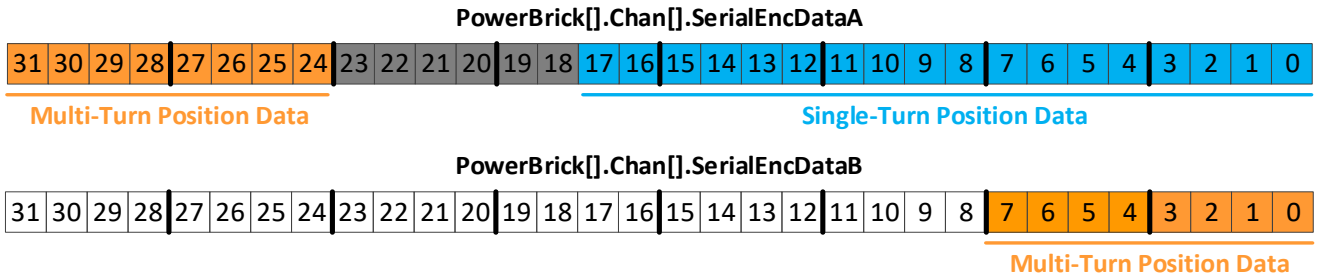
Note

Encoders with multi-turn position data are typically set up as signed.

Note

Gray code conversion should be omitted here if it had been already implemented in **PowerBrick[].Chan[].SerialEncCmd**.

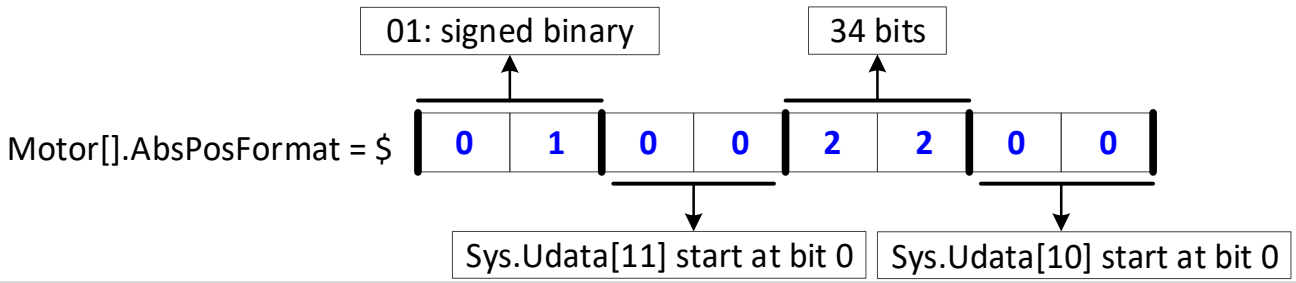
Example 8: A 34-bit binary serial encoder (for example, Panasonic) with 18 bits of single-turn and 16 bits of multi-turn position data in the following fields:



The automatic settings are not suitable for the discontinuity between the single-turn and multi-turn data.
We will assemble the absolute position word manually (in a background or initialization PLC), and hold the data in two consecutive open memory registers to feed the automatic settings. Below is the example PLC for performing this operation:

```
GLOBAL Enc1StData
GLOBAL Enc1MtData
GLOBAL Enc1Data
#define Enc1AbsPos1 Sys.Udata[10]
#define Enc1AbsPos2 Sys.Udata[11]

OPEN PLC PanasonicAbsPosPLC
LOCAL Enc1MtDataA, Enc1MtDataB;
Enc1StData = PowerBrick[0].Chan[0].SerialEncDataA & $3FFFF
Enc1MtDataA = (PowerBrick[0].Chan[0].SerialEncDataA & $FF000000) >> 24
Enc1MtDataB = PowerBrick[0].Chan[0].SerialEncDataB & $000000FF
Enc1MtData = Enc1MtDataA + Enc1MtDataB * EXP2(8)
IF (Enc1MtData > EXP2(15))  // NEGATIVE?
{
   Enc1Data = (Enc1StData - EXP2(18)) + (Enc1MtData - EXP2(16)) << 18
}
ELSE                        // POSITIVE?
{
   Enc1Data = Enc1StData + Enc1MtData * << 18
}
Enc1AbsPos1 = Enc1Data & $FFFFFFFF
Enc1AbsPos2 = (Enc1Data >> 32) & $3
DISABLE PLC PanasonicAbsPosPLC
CLOSE
```

The automatic settings can now be set up to read the absolute position at the first corresponding user defined register:




```
Motor[1].pAbsPos = Sys.Udata[10].a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $01002200
Motor[1].HomeOffset = 0
```

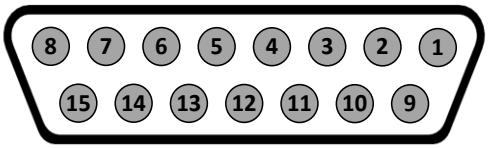
Once the example code in the sample PLC is executed, an **HMZ** command can be issued for an absolute position read.

Serial Encoders with ACC-84B

In addition to the serial encoder protocols built into DSPGate3, the Power Brick LV can accept a variety of additional protocols. These protocols are enabled by the ACC-84B. Each set of four encoders can only be programmed for one protocol at a time. This section discusses the configuration of these serial encoders.

If options M and/or N is non-zero an ACC-84B is present with a protocol indicated by the value.

																M	N		
P	B	L		-	A			0			-								0

X1-X8: D-sub DA-15F Mating: D-sub DA-15M										
Pin#	**Symbol**	**Function**	**SSI EnDat**	**Yaskawa Sigma III/V/VII**	**Tamagawa**	**Panasonic**	**Mitutoyo/ Mitsubishi**	**Biss B/C**	**Matsushita**	**OMRON 1S**
1	-	-	-		-	-				
2	-	-	-		-	-				
3	ENA –	Output	-		SEN-	-				
4	**ENCPWR**	**Output**	**Encoder Power 5 VDC (max 250 mA per channel)**							
5	DATA –	In / Out	DAT-	SDI	SD-	PS-	MRR	SLO-	/Rx	SDI
6	CLOCK –	Output	CLK-	-				MA-	-	
7	2.5V	Output	2.5 VDC – Reference							
8	PTC	Input	Motor Thermal Input							
9	-	-	-							
10	-	-	-							
11	ENA +	Output	-		SENA	-				
12	**GND**	**Common**	**Common Ground**							
13	CLOCK +	Output	CLK+	-				MA+	-	
14	DATA +	In / Out	DAT+	SDO	SD	PS	MR	SLO+	Rx	SDO
15	-	-	-	-	-	-	-	-	-	-

Caution

The +5 VDC encoder power is limited to ~250 mA per channel. For encoders requiring more current, the +5 VDC power can be alternately brought in externally through the +5 VDC ENC connector.

Caution

Encoders requiring a voltage level other than +5 VDC (higher or lower) should be powered directly from an external power supply.

Note

Quadrature / sinusoidal encoders can be wired and processed simultaneously with serial encoders on the same channel.

Pins #5, 6, 13, and 14 of the encoder feedback connectors (X1 – X8) share multiple functions: only one of these functions (per channel) can be used – configured in software – at one time:

- Hall sensor inputs (default configuration).
- Pulse and direction PFM output signals (enable using **PowerBrick[].Chan[].OutFlagD**).
- Serial encoder inputs (enable using **PowerBrick[].SerialEncEna**).
- Serial encoder inputs (enable using bit 10 of **ACC84B[].SerialEncCmd** with ACC-84B).
- Quadrature encoder inputs (serial encoder input must be disabled).
- Alternate Sinusoidal encoder inputs (with sinusoidal encoder option).

Note

Each channel is independent of the other channels and can have its own use for these pins.

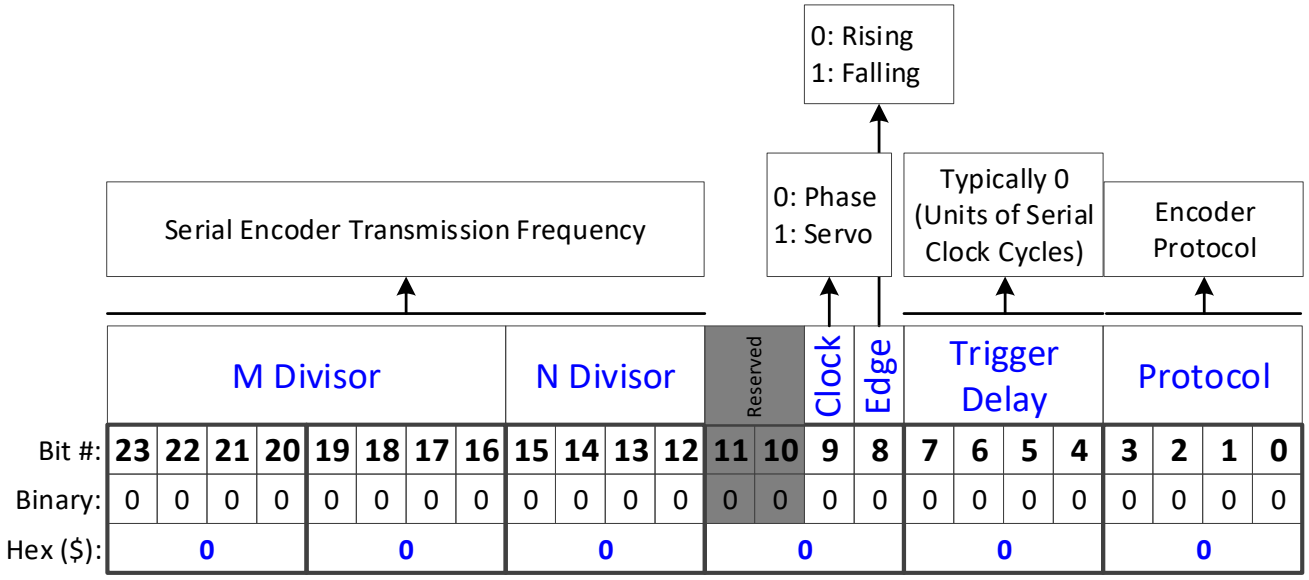
Configuring a serial encoder requires the programming of two essential structure elements.

- The Serial Encoder Control word, **ACC84B[].SerialEncCtrl**
- The Serial Encoder Command word, **ACC84B[].Chan[].SerialEncCmd**

Serial Encoder Control with ACC-84B

The Serial Encoder Control is a 24-bit, 4-channel (1 – 4, or 5 – 8), structure element. It specifies the protocol type, delay compensation time, trigger edge, trigger clock, and transmission frequency of the 4 serial encoder channels.

Channel	Serial Encoder Control Elements
1 – 4	ACC84B[0].SerialEncCtrl
5 – 8	ACC84B[1].SerialEncCtrl



Bits [23 – 12] specify the serial interface transmission frequency. This frequency (or range) is usually specified by the encoder manufacturer and programmed by the user or pre-defined by the protocol.

Bit 9 specifies the trigger source; Phase clock is recommended (value 0).

Bit 8 specifies the active edge; rising edge is recommended (value 0).

Bits [7 – 4] specify the trigger delay (in units of serial clock cycles).

Bits [3 – 0] specify the encoder protocol of the serial encoder:

Protocol	Value
–	0
–	1
SSI	2
EnDat	3

Protocol	Value
–	4
–	5
Sigma II/III/V	6
Tamagawa	7

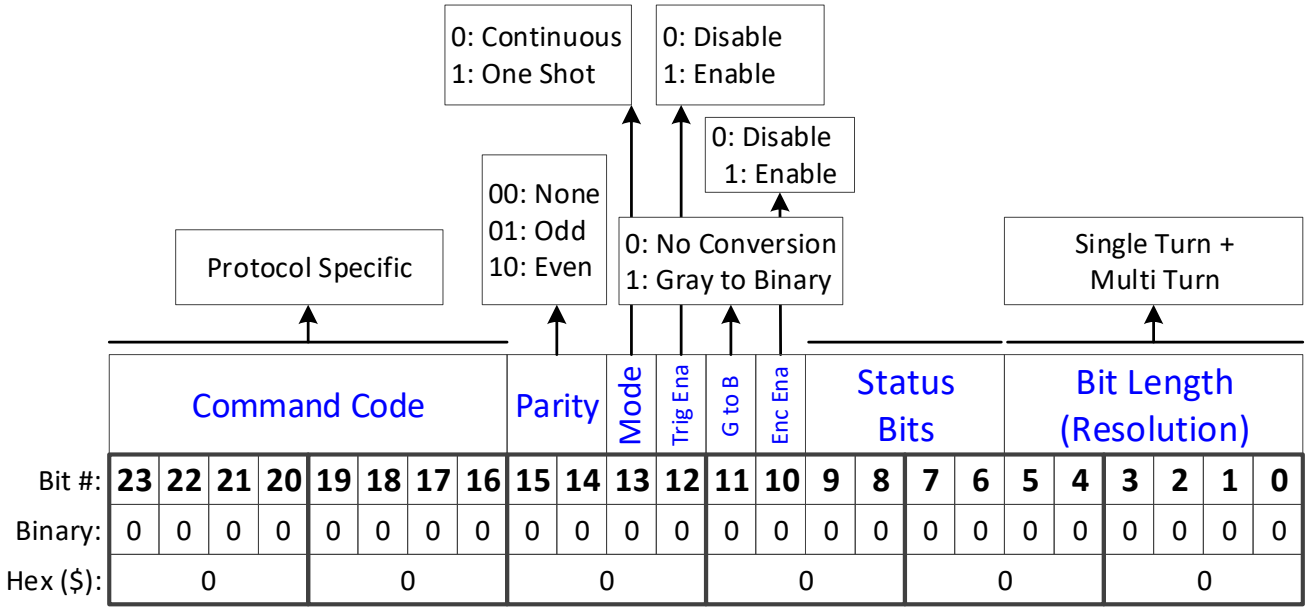
Protocol	Value
Panasonic	8
Mitutoyo	9
–	10 ($A)
Biss-B/C	11 ($B)

Protocol	Value
Matsushita	12 ($C)
Mitsubishi	13 ($D)
1S	14 ($E)
–	15 ($F)

Serial Encoder Command with ACC-84B

The Serial Encoder Command is a 24-bit, channel specific, structure element. It specifies the bit length (resolution), status bits, data type, conversion method, trigger enable, trigger mode, parity, and command code of the serial encoder channel.

Ch.#	Serial Encoder Command	Ch. #	Serial Encoder Command
1	ACC84B[0].Chan[0].SerialEncCmd	5	ACC84B[1].Chan[0].SerialEncCmd
2	ACC84B[0].Chan[1].SerialEncCmd	6	ACC84B[1].Chan[1].SerialEncCmd
3	ACC84B[0].Chan[2].SerialEncCmd	7	ACC84B[1].Chan[2].SerialEncCmd
4	ACC84B[0].Chan[3].SerialEncCmd	8	ACC84B[1].Chan[3].SerialEncCmd



Bits [23 – 16] specify the command code. This field is protocol specific.

Bits [15 – 14] specify the parity type to be expected for the received data packet (for those protocols that support parity checking).

Bit 13 specifies the trigger mode.

Bit 12 is the trigger enable toggle.

Bit 11 enables Gray code to binary conversion. This field is protocol specific.

Bit 10 is the data ready bit when read. When written it will be the serial circuitry enable bit.

Bits [9 – 6] specify the encoder status field. This field is protocol specific.

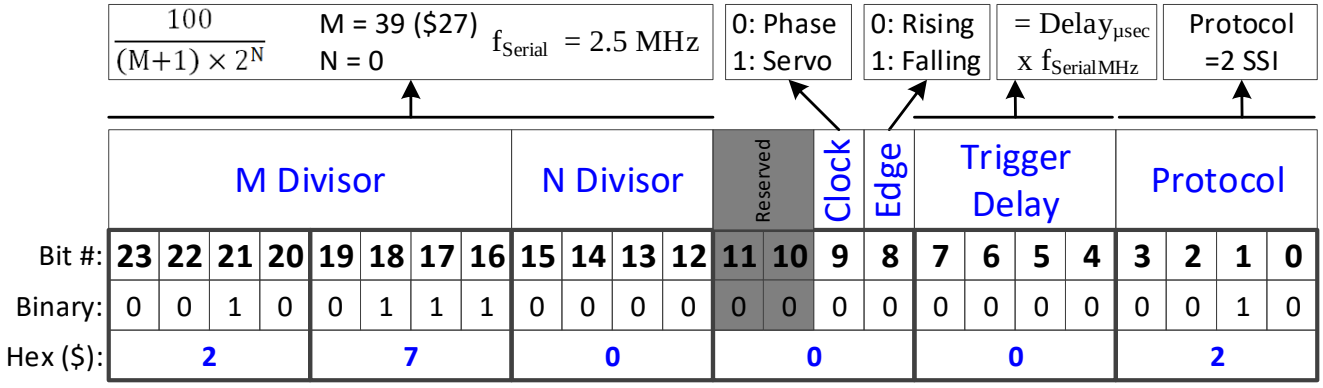
Bits [5 – 0] specify the serial encoder bit length (single-turn + multi-turn). Note that, Bit length is only required for SSI, EnDat, and BiSS.

Following, are examples for setting up the control and command words for each of the supported protocols. Also, the resulting data registers and their format.

SSI Configuration Example with ACC-84B

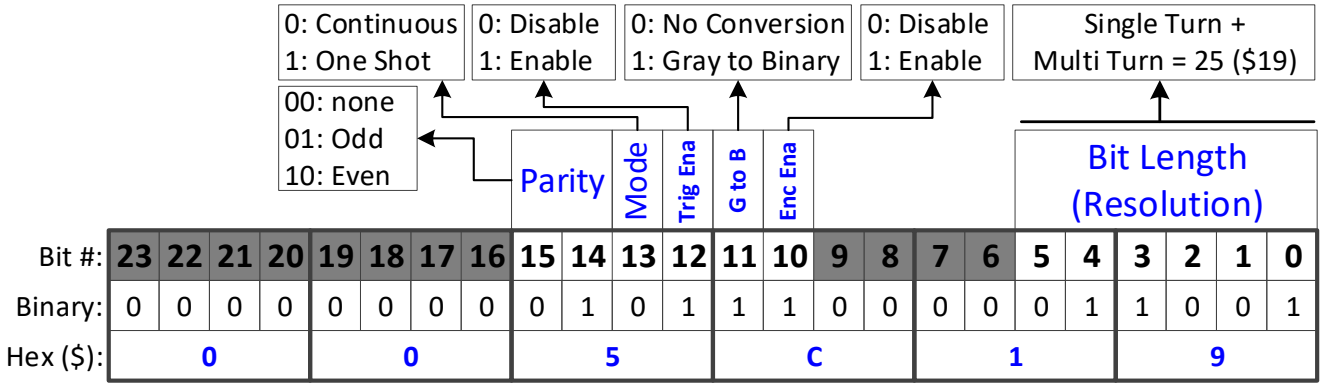
➢ **SERIAL ENCODER CONTROL – SSI**

No trigger delay, rising edge of phase, and 2.0 MHz transmission



➢ **SERIAL ENCODER COMMAND – SSI**

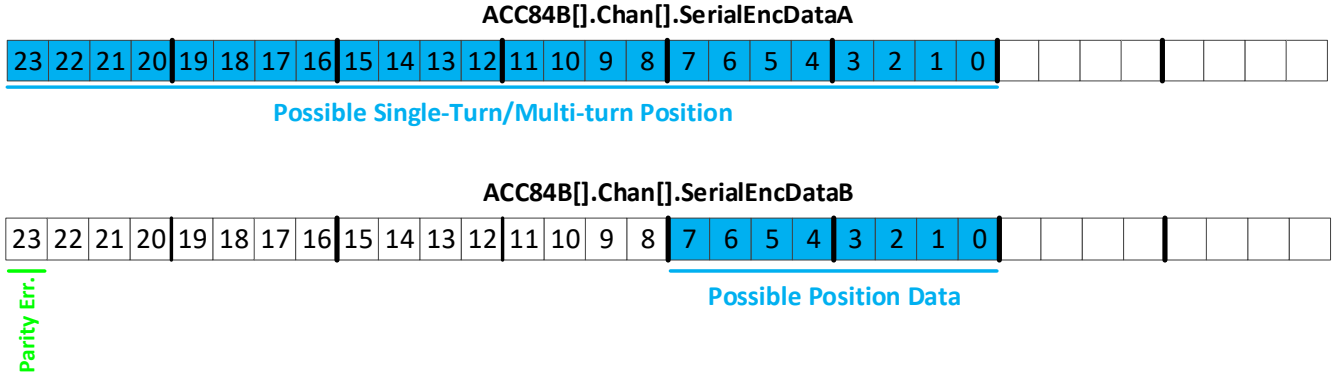
A 25-bit SSI encoder in Gray code, with odd parity




```
ACC84[0].SerialEncCtrl = $270002
ACC84[0].Chan[0].SerialEncCmd = $005C19
```

➢ **SERIAL DATA REGISTERS – SSI**

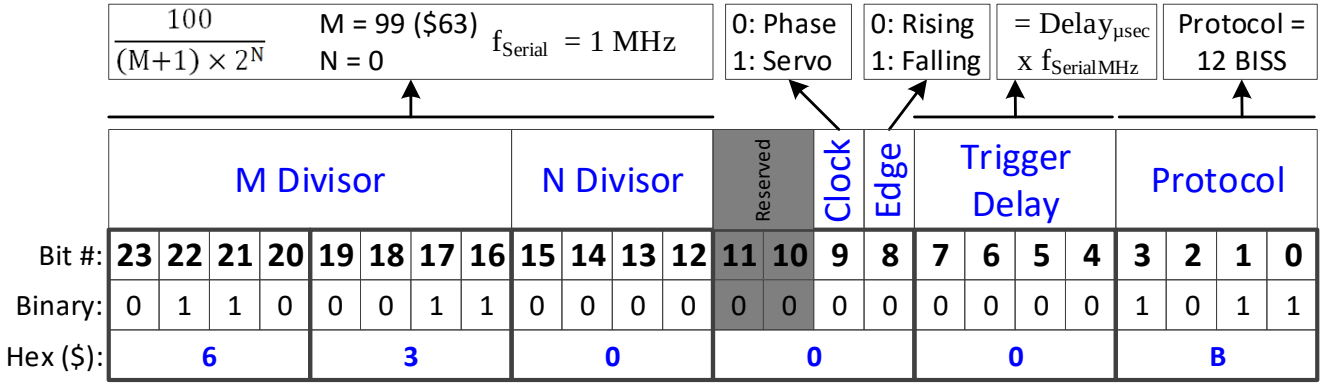
The resulting position data, status, and error bits for SSI are found in the following Serial Data Registers:



BISS B/C Configuration Example with ACC-84B

➢ **SERIAL ENCODER CONTROL EXAMPLE – BISS B/C**

No trigger delay, rising edge of phase, and 1 MHz transmission



➢ **SERIAL ENCODER COMMAND EXAMPLE – BISS C**

For the BiSS-B/C protocols, the Command Code specifies the CRC polynomial used for error detection. It must be set up to match the polynomial used for the particular BiSS encoder.

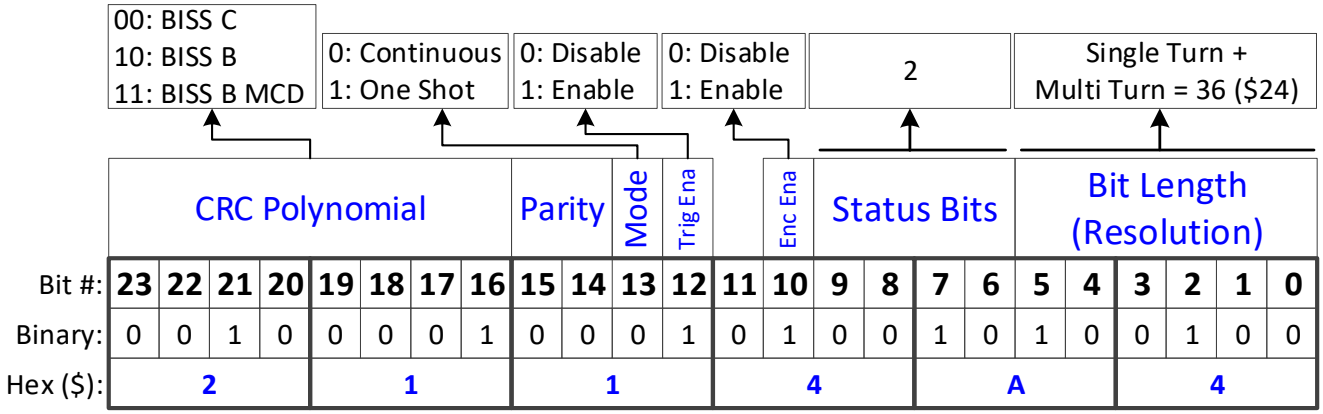
The mask bits M7 to M0 represent the coefficients for the terms x^8 to x^1, respectively, in the CRC polynomial:

$M_7x^8 + M_6x^7 + M_5x^6 + M_4x^5 + M_3x^4 + M_2x^3 + M_1x^2 + M_0x^1 + 1$

If the encoder uses a standard CRC polynomial of $x^6 + x^1 + 1$ (as with the Renishaw ResoluteTM encoders), the CRC mask value M should be set to $21.

For the BiSS protocol, Parity is used to distinguish between the BiSS-B and BiSS-C protocol variants. Bit 1 of the component is set to 0 for BiSS-C, and to 1 for BiSS-B. Bit 0 of the component is only used for BiSS-B. If it is set to 1, it permits the acceptance of a "Multi-Cycle Data" (MCD) bit from the encoder.

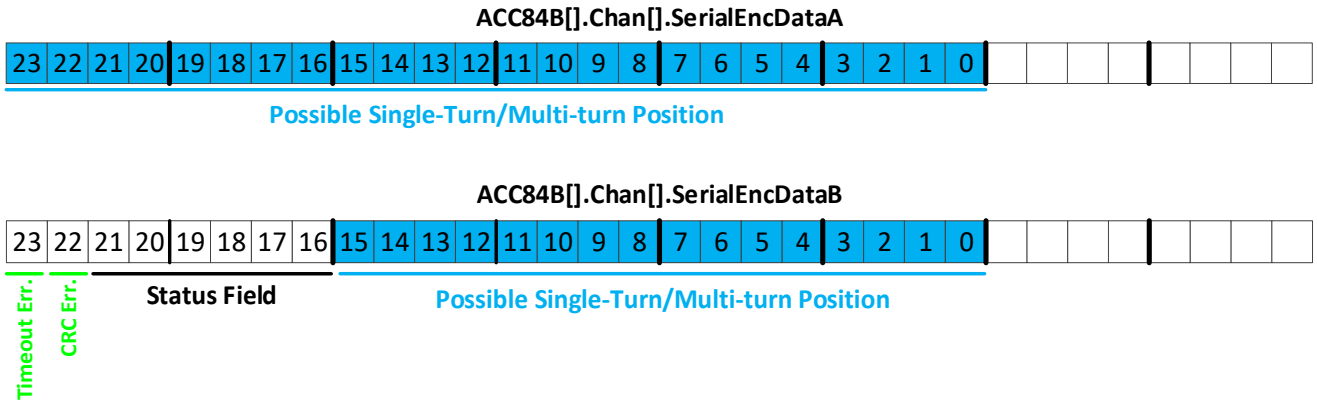
A 36-bit BISS C encoder in binary, with 2 status bits and a standard CRC polynomial.




```
ACC84[0].SerialEncCtrl = $63000B
ACC84[0].Chan[0].SerialEncCmd = $2114A4
```

➢ **SERIAL DATA REGISTERS – BISS B/C**

The resulting position data, status, and error bits for SSI are found in the following Serial Data Registers:

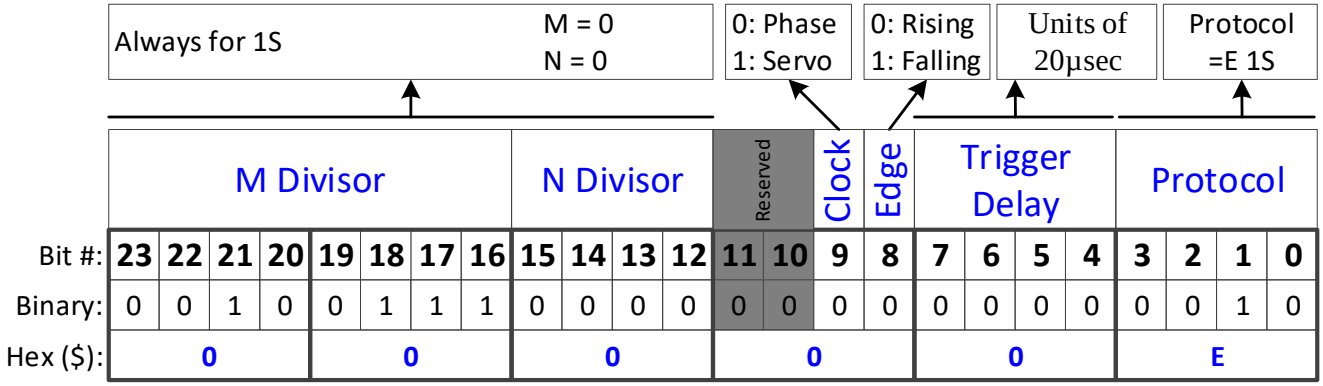


Bit #	Status Field (Renishaw Specific)
16	Indicates that the encoder scale should be cleaned. (active low)
17	Absolute position data not valid or temperature too high. (active low)

1S Configuration Example with ACC-84B

➢ **SERIAL ENCODER CONTROL – 1S**

No trigger delay, rising edge of phase.

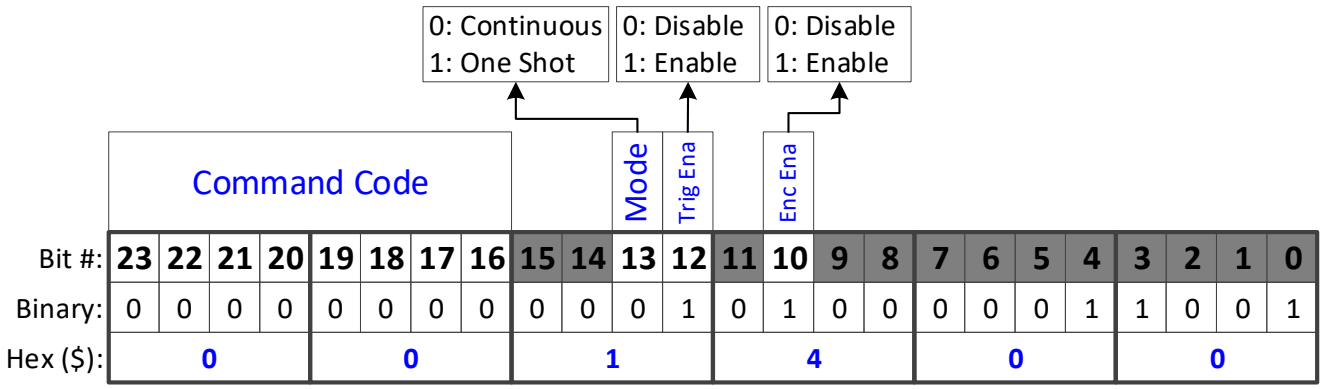


➢ **SERIAL ENCODER COMMAND – 1S**

The ACC-84B interface to a 1S encoder supports the following command codes.

- 00000000 ($00) for reporting full 40 bit absolute position.
- 00001000 ($08) for reporting low 24 bits absolute of position.
- 11011000 ($D8) for reporting low 24 bits absolute of position and alarm bits.
- 11101000 ($E8) for reporting low 24 bits absolute of position and temperature.
- 01000000 ($40) for clearing status bits.*
- 01001000 ($48) for clearing alarm bits.*
- 01011000 ($58) for setting encoder id to zero.*

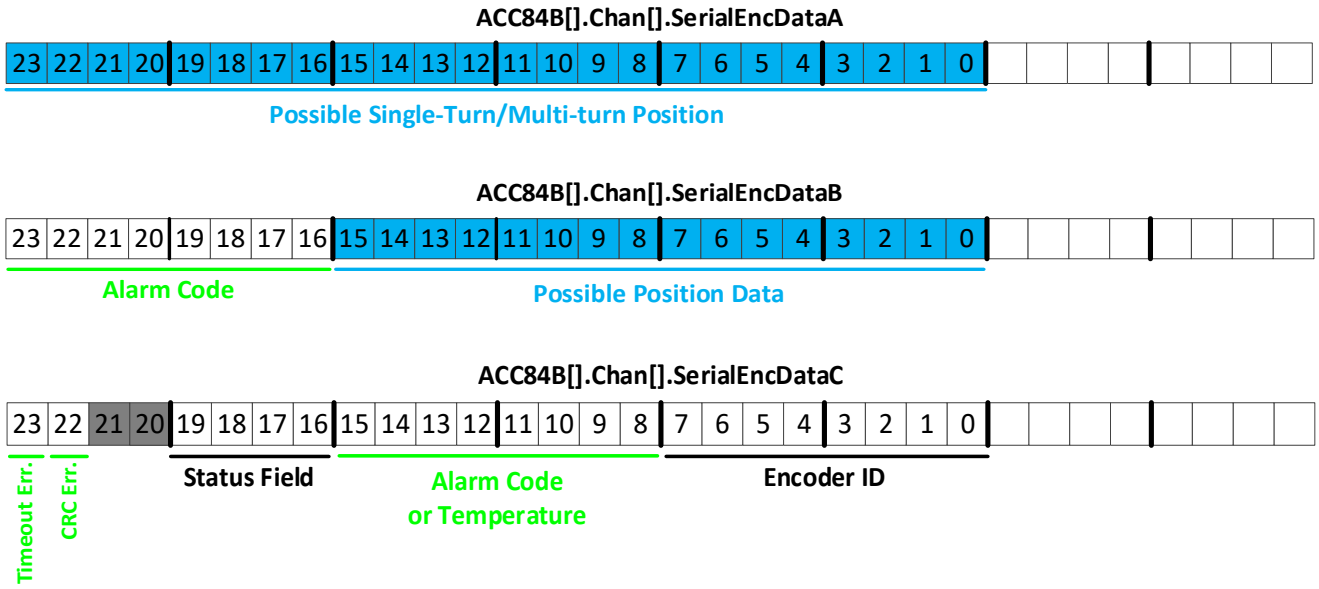
* Do not issue the last 3 command codes while the motor is enabled or the encoder data is needed, position will stop updating. The commands must be issues 8 times in a row with the trigger mode set to one shot.




```
ACC84[0].SerialEncCtrl = $E
ACC84[0].Chan[0].SerialEncCmd = $001400
```

➢ **SERIAL DATA REGISTERS – 1S**

The resulting position data, status, and error bits for 1S are found in the following Serial Data Registers:



Bit #	Alarm Code (SerialEndDataB)
16	Low Battery warning
17	Dead Battery
19	Over Speed error
20	EEPROM error
21	Position Mismatch
22	Position Mismatch Greater Than Single Turn
23	Busy error

Bit #	Alarm Code (SerialEndDataC)
08	EEPROM Busy error
09	Over Temperature error

Note

Alarm codes and temperature are only present when the correct command code is entered.

Bit #	Status Field
24	Busy or EEPROM Busy
25	Low Battery warning
26	Over Speed, Over Temperature, or EEPROM Busy
27	Position Mismatch or Dead Battery

Serial Encoder Ongoing Position Setup with ACC-84B

For the on-going "incremental" position data, it is sufficient to process whatever position data (single-turn and/or multi-turn) is available in **Acc84B[].Chan[].SerialEncDataA**. The PMAC firmware does not require processing the entire bit length, the difference change in between servo cycles is used to compute the on-going position. This will not limit the resolution or hinder the performance. Some people may choose to use strictly the single-turn data in the Encoder Conversion Table for simplicity.

A key step is to make sure that unwanted data has been cleared and the Most Significant Bit (MSB) of the data chosen is left-shifted to bit #31 in order to handle the rollover gracefully. **EncTable[].index2** is set to the number of unwanted bits to the right of the desired data, so that a right shift can be perfomed to clear that unwanted data. **EncTable[].index1** is then set to the number of bits the data must be shifted left (after the right shift) to make the (MSB) of your position data bit #31.

The following settings are required to read on-going position in counts. These settings depend primarily on the location of the position data in the **SerialEncDataA**.

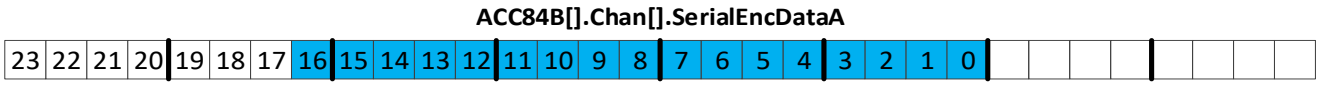
Structure Element	**Value**
EncTable[].type	1
EncTable[].pEnc	**Acc84B[].Chan[].SerialEncDataA.a**
EncTable[].pEnc1	**Sys.Pushm**
EncTable[].index1	Number of bits to left shift (second operation)
EncTable[].index2	Number of bits to right shift (first operation)
EncTable[].index3	0
EncTable[].index4	0
EncTable[].index5	0
EncTable[].index6	0
EncTable[].ScaleFactor	$1 / 2^{\textbf{EncTable[].index1}}$

Structure Element	**Value**
Motor[].ServoCtrl	1
Motor[].pEnc	**EncTable[].a**
Motor[].pEnc2	**EncTable[].a**

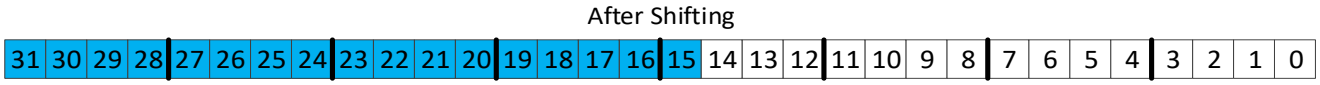
The following are examples for setting up the Encoder Conversion Table (ECT) for on-going position of various serial encoders.

Although data may appear to start at bit 0 in the script environment, internally it is only 24 bits starting at bit 8. This means data should be right shifted 8 bits more than would be expected from viewing **Acc84B[].Chan[].SerialEncDataA** in the watch window or terminal.

Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA**.



The position data should be first shifted 8 bits to the right (using **index2**) to eliminate the 8 internal bits of unwanted data. Next, the result is shifted 15 bits left (using **index1**) so that the Most Significant Bit (MSB) is at bit #31 to handle the rollover gracefully. Finally, the scale factor should reflect the new location of the Least Significant Bit (LSB).



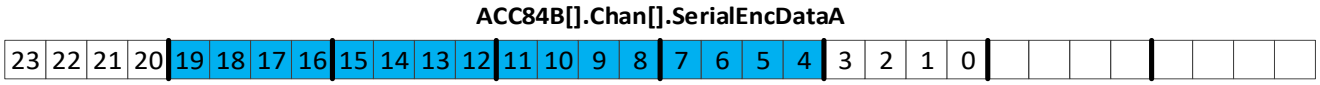

```
EncTable[1].type = 1
EncTable[1].pEnc = ACC84B[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 15
EncTable[1].index2 = 8
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].index6 = 0
EncTable[1].ScaleFactor = 1 / EXP2(15)
```

The settings below are sufficient to view motor position in the position window, in counts.

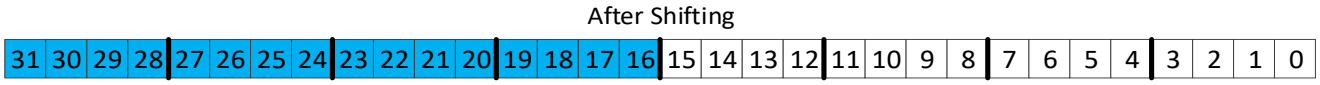
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{SingleTurn} = 2^{17} = 131{,}072$ counts per revolution for a rotary encoder. And 1/0.001 = 1,000 counts per mm for a linear encoder.

Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of the 24 bit **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.



The position data should be first shifted 12 bits to the right (using **index2**) to eliminate the 8 internal bits of unwanted data as well as the 4 bits of unwanted data sent by the encoder. Next, the result is shifted 16 bits to the left (using **index1**), so that the (MSB) is at bit #31 to handle the rollover gracefully. Finally, the scale factor should reflect the new location of the (LSB).



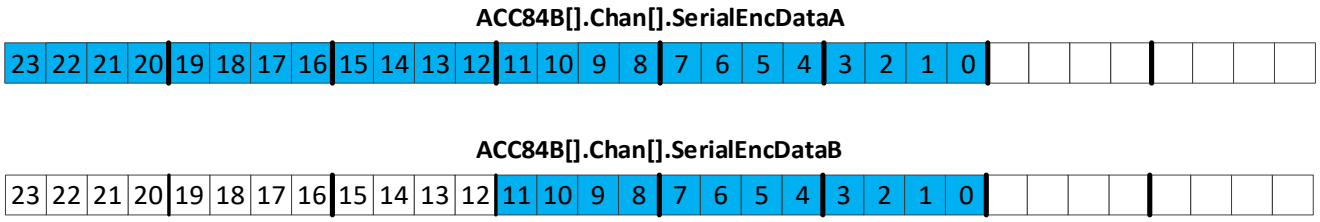

```
EncTable[1].type = 1
EncTable[1].pEnc = ACC84B[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 16
EncTable[1].index2 = 12
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1 / EXP2(16)
```

The settings below are sufficient to view motor position in the position window, in counts.

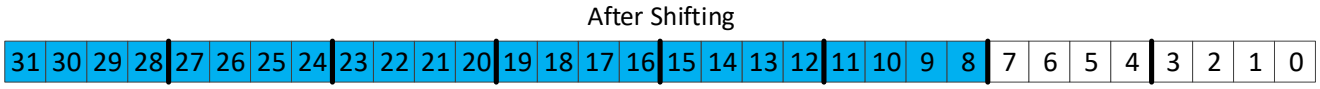
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{\text{SingleTurn}} = 2^{16}$ = 65,536 counts per revolution for a rotary encoder. And 1/0.001 = 1,000 counts per mm for a linear encoder.

Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA** and extending to bit #11 of **SerialEncDataB**.



Reading and processing the 24 bits of position data in **SerialEncDataA** is sufficient for producing the proper ongoing position. The position data should be first shifted 8 bits to the right (using **index2**) to eliminate the 8 internal bits of unwanted data. Next, the result is shifted 8 bits to the left (using **index1**), so that the (MSB) is at bit #31 to handle the rollover gracefully. Also, the scale factor should reflect the location of the (LSB).



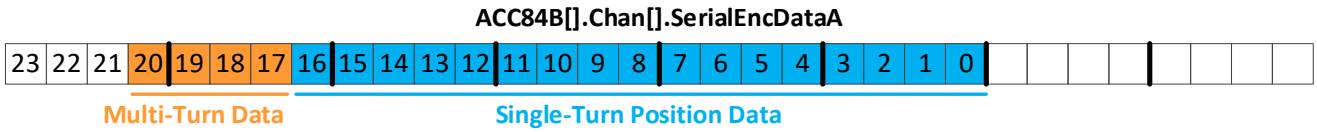

```
EncTable[1].type = 1
EncTable[1].pEnc = ACC84B[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 8
EncTable[1].index2 = 8
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1 / EXP2(8)
```

The settings below are sufficient to view motor position in the position window, in counts.

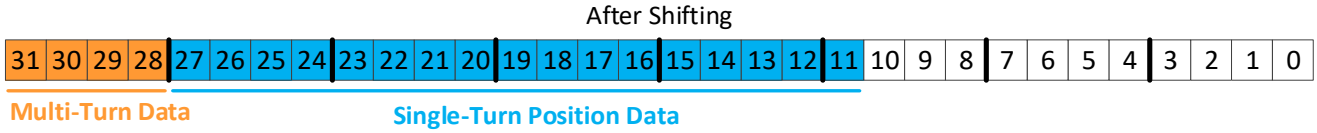
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{SingleTurn} = 2^{28} = 268,435,456$ counts per revolution for a rotary motor. And 1 / 0.000001 = 1,000,000 counts per mm for a linear motor.

Example 4: A 21-bit binary serial encoder with 17 bits of single-turn and 4 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA**.



Both single-turn and multi-turn data can be used for ongoing position. The position data should be first shifted 8 bits to the right (using **index2**) to eliminate the 8 internal bits of unwanted data. Next, the result is shifted 11 bits to the left (using **index1**), so that the (MSB) is at bit #31 to handle the rollover gracefully. Finally, the scale factor should reflect the new location of the (LSB).



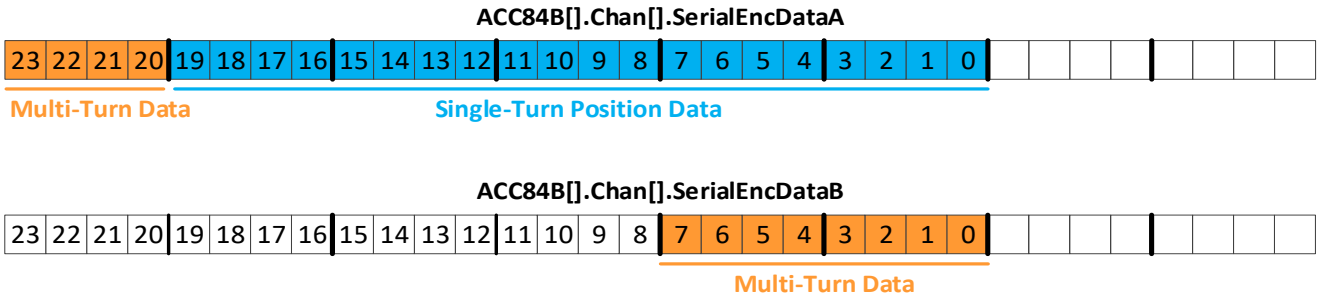

```
EncTable[1].type = 1
EncTable[1].pEnc = ACC84B[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 11
EncTable[1].index2 = 8
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].ScaleFactor = 1 / EXP2(11)
```

The settings below are sufficient to view motor position in the position window, in counts.

```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

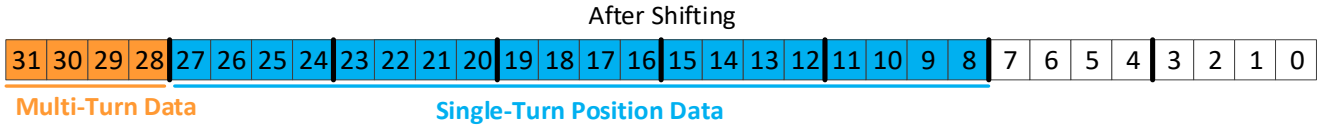
In this case, the user will see $2^{SingleTurn} = 2^{17} = 131{,}072$ counts per revolution.

Example 5: A 32-bit binary serial encoder with 20 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA** and continuously extending to bit #7 of **SerialEncDataB**.



For on-going position, we are only interested in the position data residing in **SerialEncDataA**. Some people may elect to use only the single-turn data for on-going position processing. This would require shifting to the left an extra 4 bits and a different scale factor.

But, also it is possible to simply process the 24-bit portion of single and multi-turn position data in **SerialEncDataA**. The position data should be first shifted 8 bits to the right (using **index2**) to eliminate the 8 internal bits of unwanted data. Next, the result is shifted 8 bits to the left (using **index1**), so that the (MSB) is at bit #31 to handle the rollover gracefully. Also, the scale factor should reflect the location of the (LSB).




```
EncTable[1].type = 1
EncTable[1].pEnc = ACC84B[0].Chan[0].SerialEncDataA.a
EncTable[1].pEnc1 = Sys.Pushm
EncTable[1].index1 = 8
EncTable[1].index2 = 8
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 0
EncTable[1].index6 = 0
EncTable[1].ScaleFactor = 1 / EXP2(8)
```

The settings below are sufficient to view motor position in the position window, in counts.

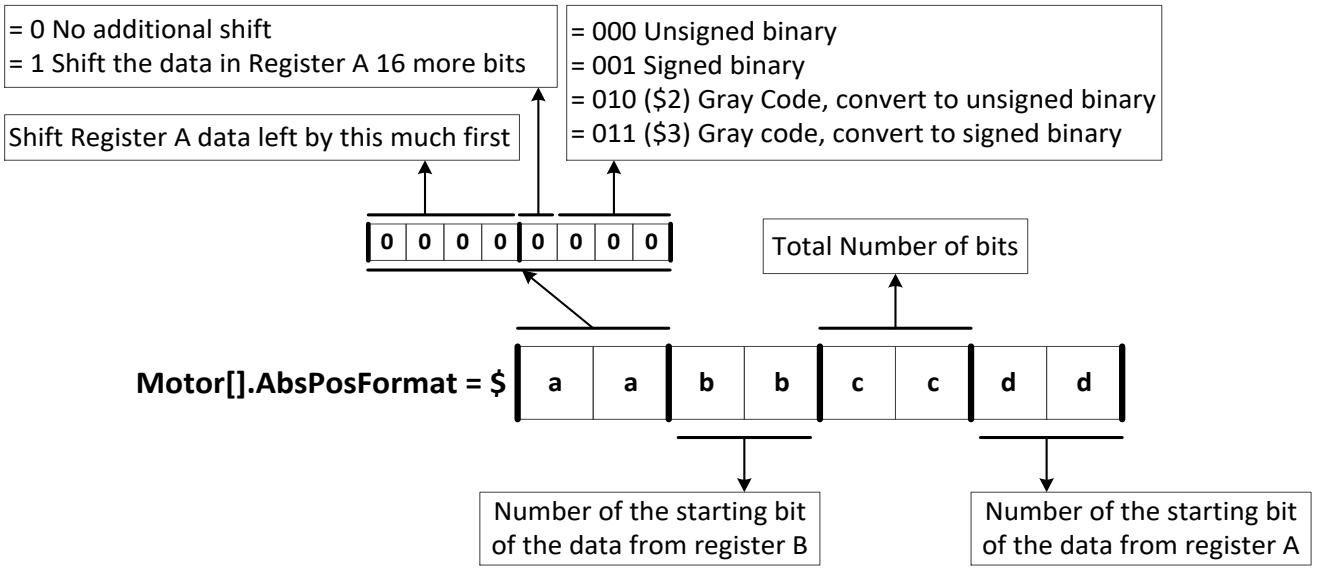
```
Motor[1].ServoCtrl = 1
Motor[1].pEnc = EncTable[1].a
Motor[1].pEnc2 = EncTable[1].a
```

In this case, the user will see $2^{SingleTurn} = 2^{20} = 1{,}048{,}576$ counts per revolution.

Serial Encoder Power-on Absolute Position Setup with ACC-84B

The absolute position is computed directly from the serial data registers, and set up using the following key structure elements:

- **Motor[].pAbsPos**, typically = **ACC84B[].Chan[].SerialEncDataA.a**
- **Motor[].AbsPosSf = Motor[].PosSf**
  - These settings should appear after Scaling to Engineering Units (in your motor setup file) so that **Motor[].PosSf** is already set.
- **Motor[].AbsPosFormat**:
  - Encoders with no multi-turn position data are unsigned. Rotary encoders with multi-turn position data are signed.



- **Motor[].HomeOffset = 0**

Note

Motor[].PowerOnMode bit 2 (value of 4) specifies an absolute position read on power up. Alternately, **#1HMZ** from the online terminal or a **HOMEZ 1** from a PLC can be issued to retrieve the absolute position.

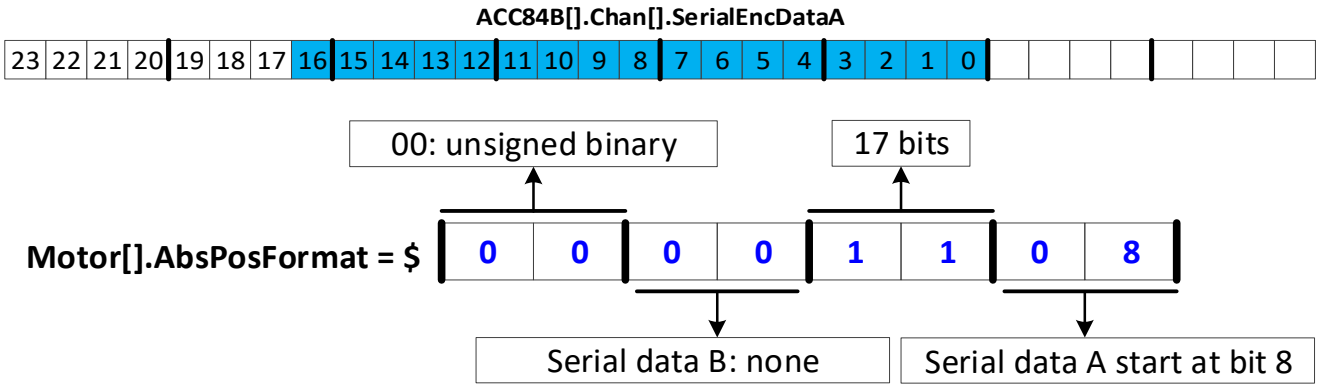
Note

Gray code conversion should be omitted here if it had been already implemented in **ACC84B[].Chan[].SerialEncCmd**.

Following, are examples for setting up the absolute position read with various serial encoders. These settings depend primarily on the location of the position data in **SerialEncDataA** and **SerialEncDataB**.

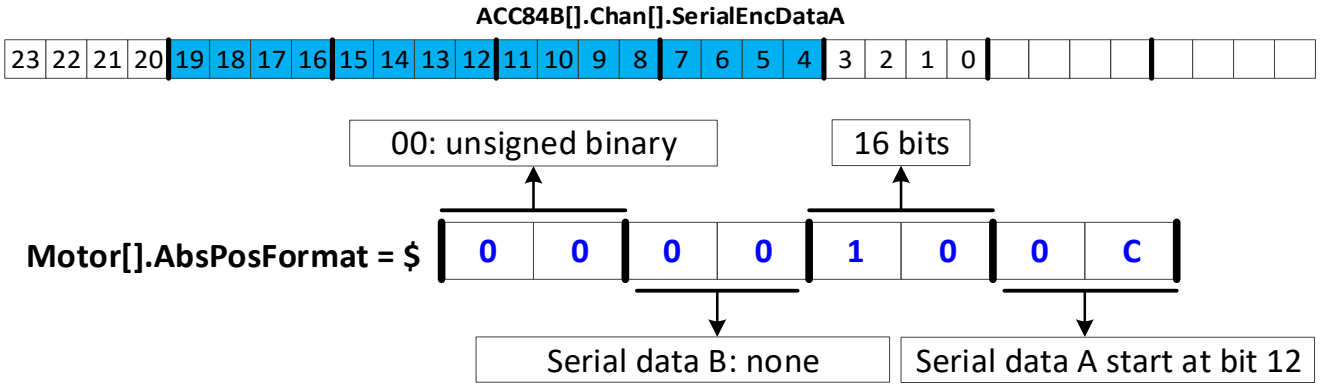
Although data may appear to start at bit 0 in the script environment, internally it is only 24 bits starting at bit 8. This means data should be right shifted 8 bits more than would be expected from viewing **Acc84B[].Chan[].SerialEncDataA** in the watch window or terminal.

Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA**.



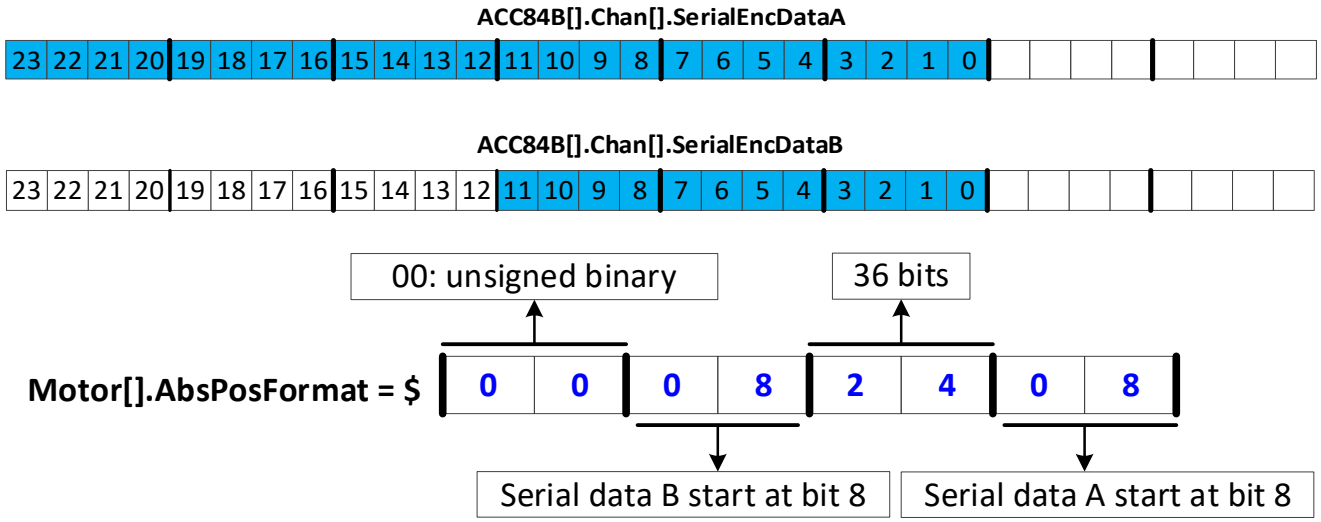

```
Motor[1].pAbsPos = ACC84B[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $00001108
Motor[1].HomeOffset = 0
```

Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of the 24 bit **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.



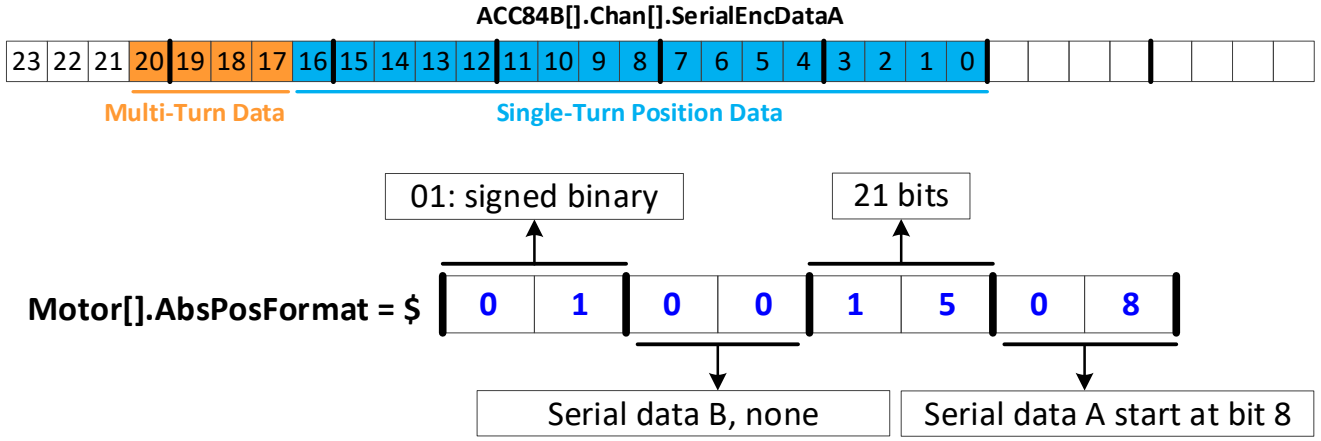

```
Motor[1].pAbsPos = ACC84B[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $0000100C
Motor[1].HomeOffset = 0
```

Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA** and extending to bit #11 of **SerialEncDataB**.



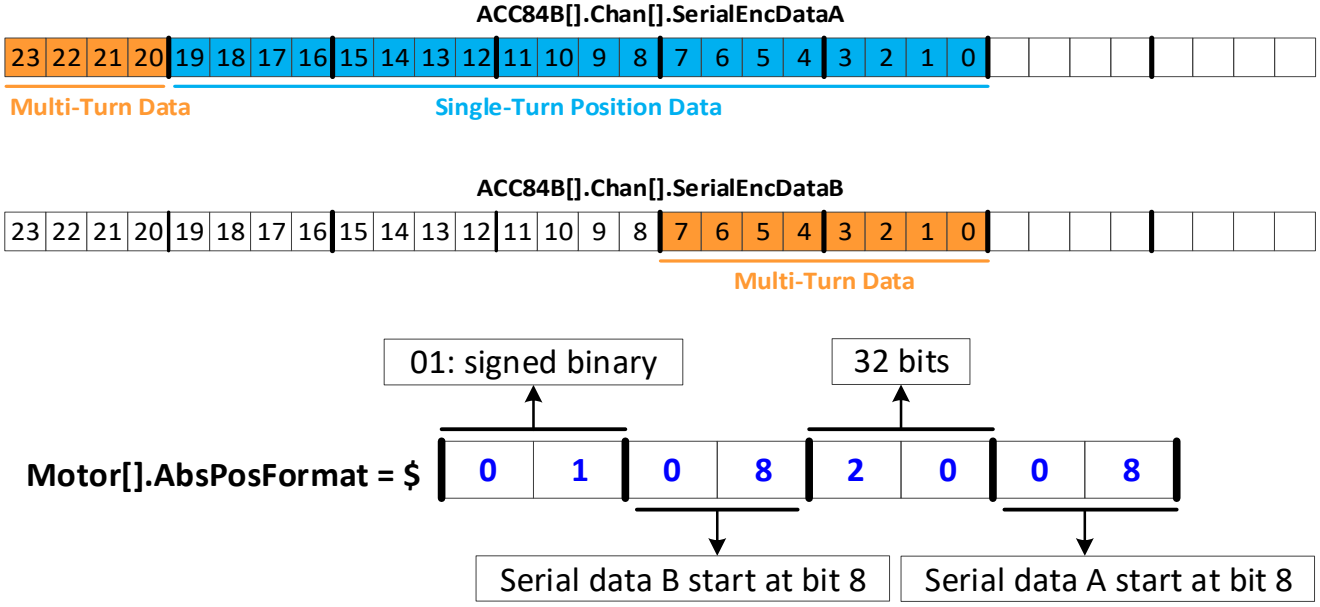

```
Motor[1].pAbsPos = ACC84B[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $00082408
Motor[1].HomeOffset = 0
```

Example 4: A 21-bit binary serial encoder with 17 bits of single-turn and 4 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA**.



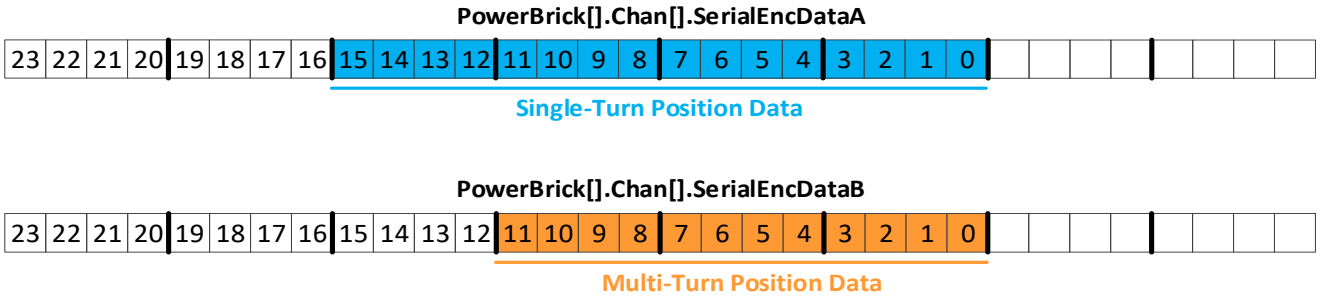

```
Motor[1].pAbsPos = ACC84B[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $01001508
Motor[1].HomeOffset = 0
```

Example 5: A 32-bit binary serial encoder with 20 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA** and continuously extending to bit #7 of **SerialEncDataB**.

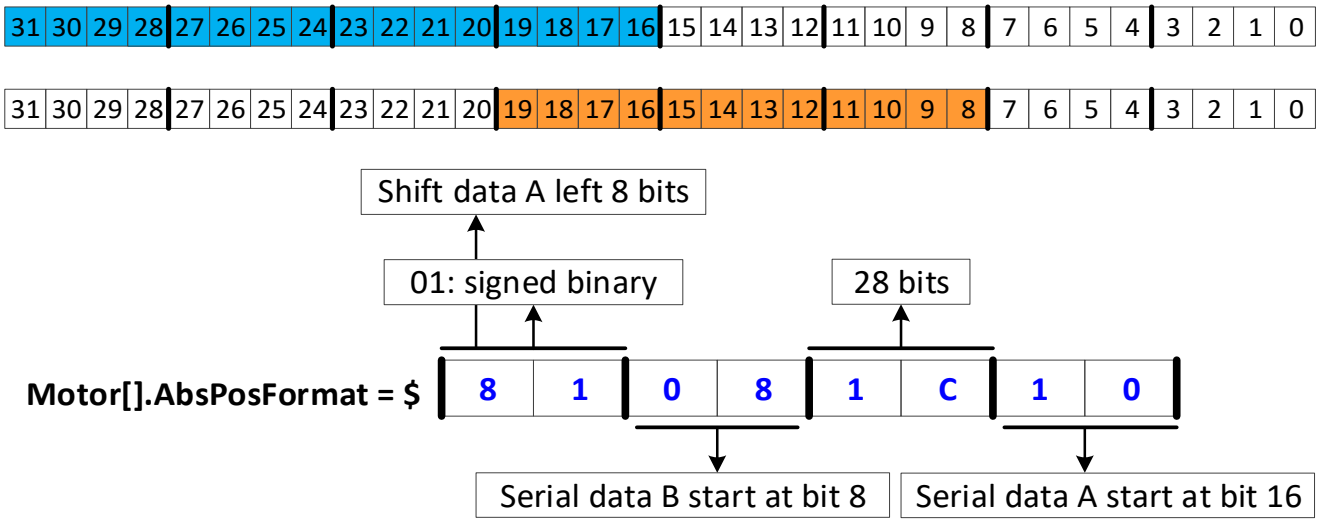



```
Motor[1].pAbsPos = ACC84B[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $01082008
Motor[1].HomeOffset = 0
```

Example 6: A 28-bit binary serial encoder with 16 bits of single-turn in the lower fields of the 24 bit **SerialEncDataA**, and 12 bits of multi-turn position data in the lower fields of the 24 bit **SerialEncDataB**.

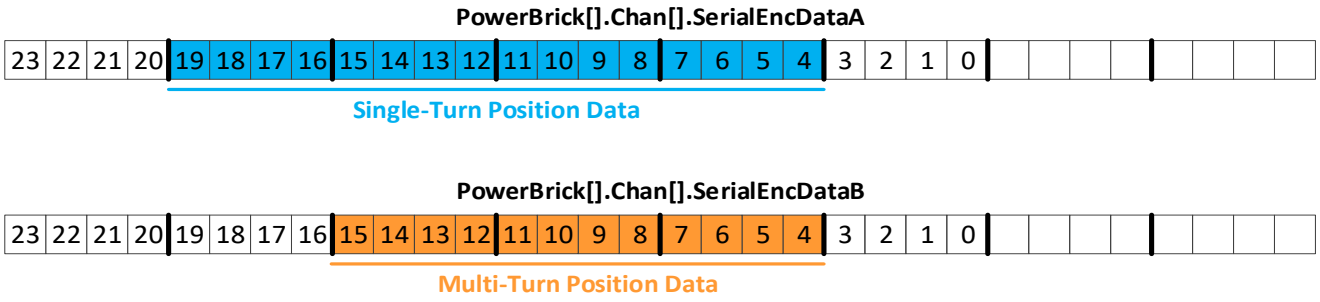


The single turn data must be shifted 8 bits left first, to make it contiguous with the multi-turn data. This shift is done using the upper 5 bits of the **$aa** byte of **Motor[].AbsPosFormat**. The data would then look like:

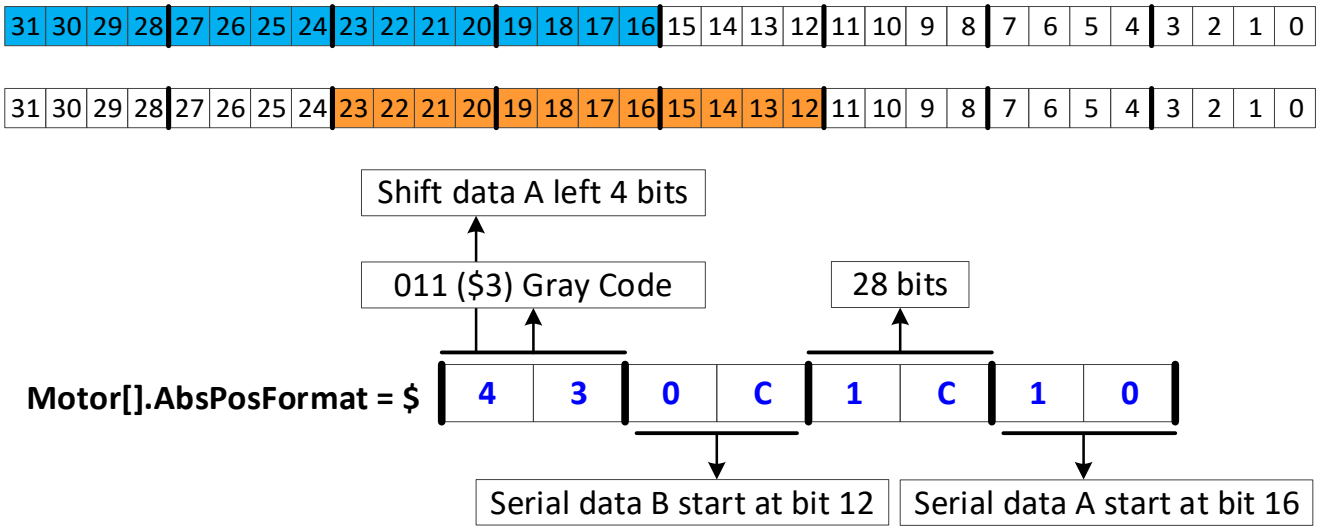



```
Motor[1].pAbsPos = ACC84B[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $81081C10
Motor[1].HomeOffset = 0
```

Example 7: A 28-bit Gray code serial encoder with 16 bits of single-turn data starting at bit #4 of the 24 bit **SerialEncDataA**, and 12 bits of multi-turn position data starting at bit #4 of the 24 bit **SerialEncDataB**.



The single turn data must be shifted 4 bits left to become contiguous with **SerialEncDataB**. This shift is done using the upper 5 bits of the **$aa** byte of **Motor[].AbsPosFormat**. The data would then look like:




```
Motor[1].pAbsPos = ACC84B[0].Chan[0].SerialEncDataA.a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $430C1C10
Motor[1].HomeOffset = 0
```

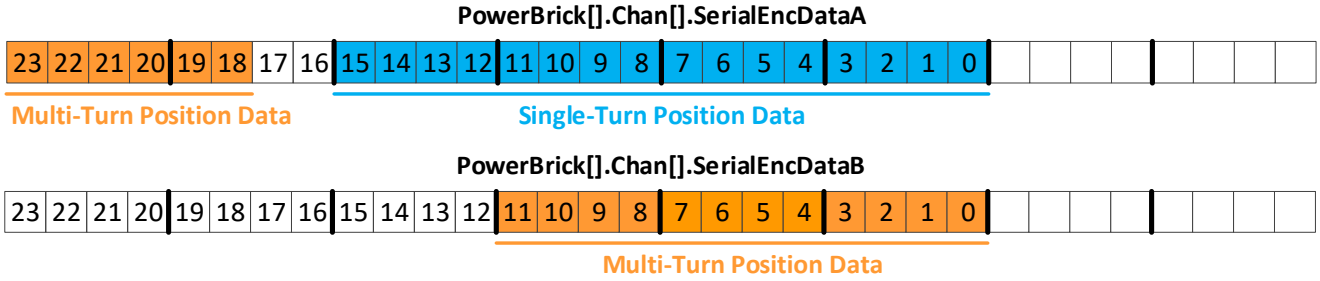
Note

Encoders with multi-turn position data are typically set up as signed.

Note

Gray code conversion should be omitted here if it had been already implemented in **PowerBrick[].Chan[].SerialEncCmd**.

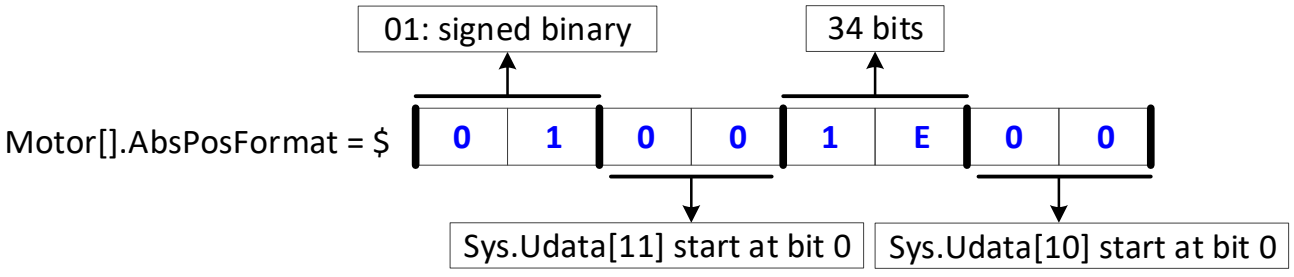
Example 8: A 34-bit binary serial encoder (for example, Panasonic) with 16 bits of single-turn and 18 bits of multi-turn position data in the following fields:



The automatic settings are not suitable for the discontinuity between the single-turn and multi-turn data.
We will assemble the absolute position word manually (in a background or initialization PLC), and hold the data in two consecutive open memory registers to feed the automatic settings. Below is the example PLC for performing this operation:

```
GLOBAL Enc1StData;
GLOBAL Enc1MtData;
GLOBAL Enc1Data;
#define Enc1AbsPos1 Sys.Udata[10]
#define Enc1AbsPos2 Sys.Udata[11]

OPEN PLC PanasonicAbsPosPLC
LOCAL Enc1MtDataA, Enc1MtDataB;
Enc1StData  = (PowerBrick[0].Chan[0].SerialEncDataA & $00FFFF00) >> 8
Enc1MtDataA = (PowerBrick[0].Chan[0].SerialEncDataA & $FC000000) >> 26
Enc1MtDataB = (PowerBrick[0].Chan[0].SerialEncDataB & $000FFF00) >>8
Enc1MtData = Enc1MtDataA + Enc1MtDataB << 6
IF (Enc1MtData > EXP2(17))  // NEGATIVE?
{
   Enc1Data = (Enc1StData - EXP2(16)) + (Enc1MtData – EXP2(18)) << 16
}
ELSE                        // POSITIVE?
{
   Enc1Data = Enc1StData + Enc1MtData << 16
}
Enc1AbsPos1 = Enc1Data & $FFFFFFFF
Enc1AbsPos2 = (Enc1Data >> 32) & $3
DISABLE PLC PanasonicAbsPosPLC
CLOSE
```

The automatic settings can now be set up to read the absolute position at the first corresponding user defined register:




```
Motor[1].pAbsPos = Sys.Udata[10].a
Motor[1].AbsPosSf = Motor[1].PosSf
Motor[1].AbsPosFormat = $01001E00
Motor[1].HomeOffset = 0
```

Once the example code in the sample PLC is executed, an **HMZ** command can be issued for an absolute position read.

XY2-100 Galvanometer Interface

For setup of XY2-100 Serial Link (also known as Serial Link 1 and XYZ-100), refer to the ACC-84E manual.

Table Based Position Compare

For setup of Table Based Position Compare, refer to the ACC-84E manual.

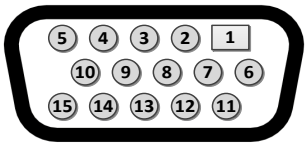
Analog I/O (X9-X12)

The features described in this section are available if option O is non-zero.

																		O	
P	B	L		-	A			0			-								0

Each of the analog I/O connectors (X9, X10, X11, and X12) provides:

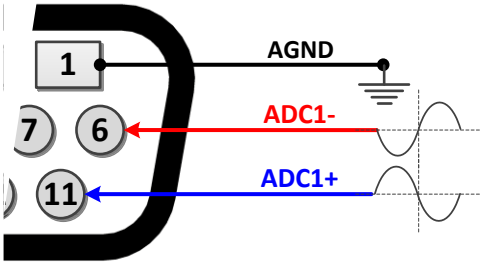
- 2 x 16-bit Analog Inputs
- 2 x ~14-bit Analog Outputs
- 2 x General Purpose Relays / Brakes
- 2 x General Purpose Inputs / External Amp Faults

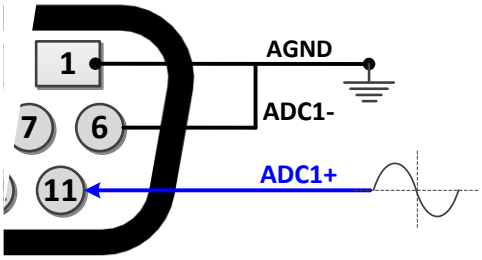
X9-X10: D-Sub DE-15 F **Mating: D-Sub DE-15 M**			
Pin #	**Symbol**	**Function**	**Description**
1	AGND	Ground	Common Analog Ground
2	DAC1-	Output	Analog Output 1-
3	AE-NO1	Relay	Normally Open GP Relay / Brake 1
4	ADC2+	Input	Analog Input 2+
5	AE-COM2	Common	GP Relay / Brake Common 2
6	ADC1-	Input	Analog Input 1-
7	DAC1+	Output	Analog Output 1+
8	AMPFLT1	Input	GP Input / Ext Amp Fault 1
9	DAC2-	Output	Analog Output 2-
10	AE-NO2	Relay	Normally Open GP Relay / Brake 2
11	ADC1+	Input	Analog Input 1+
12	AE-COM1	Common	GP Relay / Brake Common 1
13	ADC2-	Input	Analog Input 2-
14	DAC2+	Output	Analog Output 2+
15	AMPFLT2	Input	GP / Amp Fault Input 2

Setting up the Analog (ADC) Inputs

The analog inputs accept ±5 V differential signals, or ±10 V single-ended signals.

➢ **DIFFERENTIAL ANALOG INPUT SIGNAL** ➢ ➢ ➢ **SINGLE ENDED ANALOG INPUT SIGNAL**





Note

For single-ended connections, tie the negative ADC pin to ground.

The ADC software data resides in the upper 16 bits of the 32-bit structure element **PowerBrick[].Chan[].AdcAmp[2]**. The structure elements do not allow bit masking (of the upper 16 bits), hence scaling (shifting) is required to obtain the raw ADC data. Using the explicit address registers makes bit masking easier:

Channel/Connector	Address	Structure Element
ADC 1, X9	$900028	PowerBrick[0].Chan[0].AdcAmp[2]
ADC 2, X9	$9000A8	PowerBrick[0].Chan[1].AdcAmp[2]
ADC 1, X10	$900128	PowerBrick[0].Chan[2].AdcAmp[2]
ADC 2, X10	$9001A8	PowerBrick[0].Chan[3].AdcAmp[2]

Channel/Connector	Address	Structure Element
ADC 1, X11	$904028	PowerBrick[1].Chan[0].AdcAmp[2]
ADC 2, X11	$9040A8	PowerBrick[1].Chan[1].AdcAmp[2]
ADC 1, X12	$904128	PowerBrick[1].Chan[2].AdcAmp[2]
ADC 2, X12	$9041A8	PowerBrick[1].Chan[3].AdcAmp[2]

Note

The explicit address register(s) can be found by subtracting **Sys.piom** from **PowerBrick[].Chan[].AdcAmp[2].a**

Note

The ADC input data must be in the "unpacked" format to be read properly; use **PowerBrick[].Chan[].PackInData = 0**.

➢ **RAW ADC DATA (BITS)**

```
Sys.WpKey = $AAAAAAAA                          // Disable Write-Protection

PowerBrick[0].Chan[0].PackInData = 0           // Unpack Input Data, ADC1 X9
PowerBrick[0].Chan[1].PackInData = 0           // Unpack Input Data, ADC2 X9
PowerBrick[0].Chan[2].PackInData = 0           // Unpack Input Data, ADC1 X10
PowerBrick[0].Chan[3].PackInData = 0           // Unpack Input Data, ADC2 X10
PowerBrick[1].Chan[0].PackInData = 0           // Unpack Input Data, ADC1 X11
PowerBrick[1].Chan[1].PackInData = 0           // Unpack Input Data, ADC2 X11
PowerBrick[1].Chan[2].PackInData = 0           // Unpack Input Data, ADC1 X12
PowerBrick[1].Chan[3].PackInData = 0           // Unpack Input Data, ADC2 X13

PTR ADC1X9 ->S.IO:$900028.16.16                // ADC1 X9  [Counts]
PTR ADC2X9 ->S.IO:$9000A8.16.16                // ADC2 X9  [Counts]
PTR ADC1X10->S.IO:$900128.16.16                // ADC1 X10 [Counts]
PTR ADC2X10->S.IO:$9001A8.16.16                // ADC2 X10 [Counts]
PTR ADC1X11->S.IO:$904028.16.16                // ADC1 X11 [Counts]
PTR ADC2X11->S.IO:$9040A8.16.16                // ADC2 X11 [Counts]
PTR ADC1X12->S.IO:$904128.16.16                // ADC1 X12 [Counts]
PTR ADC2X12->S.IO:$9041A8.16.16                // ADC2 X12 [Counts]
```

The analog inputs have 16 bits of resolution (65,536 software counts) spanning over the full range of the input voltage. Wiring ±10 V voltage in single-ended, or ±5 V in differential mode produces the following counts in software:

Single-Ended [VDC]	Differential [VDC]	Software Counts
-10	-5	-32768
0	0	0
10	5	+32768

➢ **SCALING THE ANALOG INPUT DATA**

For general purpose usage, the ADC data (reported in bits) can be easily scaled and converted into "user" voltage or units (e.g. force, height). In the example PLC below:

- The global parameter **ADCnXxxZeroOffset** represents the voltage offset with a zero volt input. This is user adjustable.
- The pointer **ADCnXxx** reports the raw ADC data in software counts, units of 16-bit (±32768).
- The global parameter **ADCnXxxVolts** reports the ADC data in "user" volts.

Where **n** is the ADC channel number (1 or 2) of the corresponding **xx** connector (X9, X10, X11, or X12).

```
GLOBAL ADC1X9Volts = 0                      // Voltage input, ADC1 X9  [volt]
GLOBAL ADC2X9Volts = 0                      // Voltage input, ADC2 X9  [volt]
GLOBAL ADC1X10Volts = 0                     // Voltage input, ADC1 X10 [volt]
GLOBAL ADC2X10Volts = 0                     // Voltage input, ADC2 X10 [volt]
GLOBAL ADC1X11Volts = 0                     // Voltage input, ADC1 X11 [volt]
GLOBAL ADC2X11Volts = 0                     // Voltage input, ADC2 X11 [volt]
GLOBAL ADC1X12Volts = 0                     // Voltage input, ADC1 X12 [volt]
GLOBAL ADC2X12Volts = 0                     // Voltage input, ADC2 X12 [volt]

GLOBAL ADC1X9ZeroOffset = 0.038             // Zero Volt Offset, ADC1 X9  [volt] --USER ADJUSTABLE
GLOBAL ADC2X9ZeroOffset = 0.038             // Zero Volt Offset, ADC2 X9  [volt] --USER ADJUSTABLE
GLOBAL ADC1X10ZeroOffset = 0.038            // Zero Volt Offset, ADC1 X10 [volt] --USER ADJUSTABLE
GLOBAL ADC2X10ZeroOffset = 0.038            // Zero Volt Offset, ADC2 X10 [volt] --USER ADJUSTABLE
GLOBAL ADC1X11ZeroOffset = 0.038            // Zero Volt Offset, ADC1 X11 [volt] --USER ADJUSTABLE
GLOBAL ADC2X11ZeroOffset = 0.038            // Zero Volt Offset, ADC2 X11 [volt] --USER ADJUSTABLE
GLOBAL ADC1X12ZeroOffset = 0.038            // Zero Volt Offset, ADC1 X12 [volt] --USER ADJUSTABLE
GLOBAL ADC2X12ZeroOffset = 0.038            // Zero Volt Offset, ADC2 X12 [volt] --USER ADJUSTABLE
```

```
OPEN PLC ExamplePLC
ADC1X9Volts  = (ADC1X9  * 10 / 32768) - ADC1X9ZeroOffset        // ADC1, X9  [volts]
ADC2X9Volts  = (ADC2X9  * 10 / 32768) - ADC2X9ZeroOffset        // ADC2, X9  [volts]
ADC1X10Volts = (ADC1X10 * 10 / 32768) - ADC1X10ZeroOffset       // ADC1, X10 [volts]
ADC2X10Volts = (ADC2X10 * 10 / 32768) - ADC2X10ZeroOffset       // ADC2, X10 [volts]
ADC1X11Volts = (ADC1X11 * 10 / 32768) - ADC1X11ZeroOffset       // ADC1, X11 [volts]
ADC2X11Volts = (ADC2X11 * 10 / 32768) - ADC2X11ZeroOffset       // ADC2, X11 [volts]
ADC1X12Volts = (ADC1X12 * 10 / 32768) - ADC1X12ZeroOffset       // ADC1, X12 [volts]
ADC2X12Volts = (ADC2X12 * 10 / 32768) - ADC2X12ZeroOffset       // ADC2, X12 [volts]
CLOSE
```

- **USING THE ADC FOR SERVO FEEDBACK**

Using the ADC data for servo feedback requires bringing it into the Encoder Conversion Table (ECT) into which the motor's position and velocity elements are assigned to.

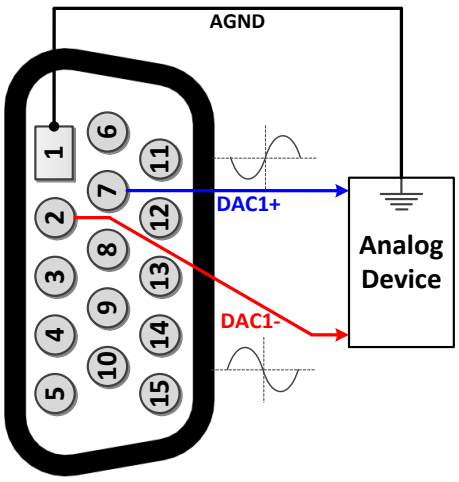
- **EXAMPLE:**

```
EncTable[9].Type = 1
EncTable[9].pEnc = PowerBrick[0].Chan[0].AdcAmp[2].a
EncTable[9].pEnc1 = Sys.pushm
EncTable[9].index1 = 16
EncTable[9].index2 = 16
EncTable[9].index3 = 0
EncTable[9].index4 = 0
EncTable[9].index5 = 0
EncTable[9].ScaleFactor = 1 / EXP2(16)

Motor[9].ServoCtrl = 1
Motor[9].pEnc = EncTable[9].a
Motor[9].pEnc2 = EncTable[9].a
```

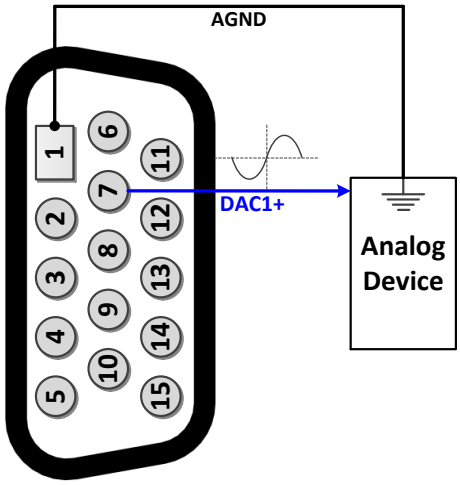
Setting up the Analog (DAC) Outputs

The analog outputs provide ±10 V signals interfacing to either differential or single-ended devices.

Differential DAC Output Signal



Single Ended DAC Output Signal



The analog output circuitry is filtered PWM optimized (in hardware) for a cut off frequency of about 15 kHz. The nominal PWM frequency of 20 kHz in the Power Brick LV produces excellent analog output signals.

Note

These analog outputs are synthesized filtered PWM. They are designed for general purpose use; they are not industrially graded for servo use. True DAC outputs are typically used in servo applications.

The analog output command data resides in the upper 16 bits of the 32-bit structure element **PowerBrick[].Chan[].Pwm[3].** The structure elements do not allow bit masking (of the upper 16 bits), hence scaling (shifting) is required to write to the outputs properly. Using the explicit address registers makes it easier for bit masking:

Channel/Connector	Address	Channel/Connector	Address
Channel 1, X9	$90004C	Channel 1, X11	$90404C
Channel 2, X9	$9000CC	Channel 2, X11	$9040CC
Channel 1, X10	$90014C	Channel 1, X12	$90414C
Channel 2, X10	$9001CC	Channel 2, X12	$9041CC

Note

The explicit address register(s) can be found by subtracting **Sys.piom** from **PowerBrick[].Chan[].Pwm[3].a**

Note

Writing directly into **PowerBrick[].Chan[].Pwm[3]** register to produce voltage output requires shifting left by 16 bits (or multiplying by 65536).

Note

The command output data must be in the "unpacked" format; **PowerBrick[].Chan[].PackOutData = 0**.

Note

This output comes out of phase D. It must be set for PWM. Therefore, bit #3 of PowerBrick[].Chan[].OutputMode must be set to 0 (default).

➢ **COMMAND REGISTER POINTERS**

```
Sys.WpKey = $AAAAAAAA                          // Disable Write-Protection

PowerBrick[0].Chan[0].PackOutData = 0          // DAC1, X9,  Unpack Output Data
PowerBrick[0].Chan[1].PackOutData = 0          // DAC2, X9,  Unpack Output Data
PowerBrick[0].Chan[2].PackOutData = 0          // DAC1, X10, Unpack Output Data
PowerBrick[0].Chan[3].PackOutData = 0          // DAC2, X10, Unpack Output Data
PowerBrick[1].Chan[0].PackOutData = 0          // DAC1, X11, Unpack Output Data
PowerBrick[1].Chan[1].PackOutData = 0          // DAC2, X11, Unpack Output Data
PowerBrick[1].Chan[2].PackOutData = 0          // DAC1, X12, Unpack Output Data
PowerBrick[1].Chan[3].PackOutData = 0          // DAC2, X12, Unpack Output Data

PTR DAC1X9-> S.IO:$90004C.16.16                // DAC Channel 1, X9    [Counts]
PTR DAC2X9-> S.IO:$9000CC.16.16                // DAC Channel 2, X9    [Counts]
PTR DAC1X10->S.IO:$90014C.16.16                // DAC Channel 1, X10   [Counts]
PTR DAC2X10->S.IO:$9001CC.16.16                // DAC Channel 2, X10   [Counts]
PTR DAC1X11->S.IO:$90404C.16.16                // DAC Channel 1, X11   [Counts]
PTR DAC2X11->S.IO:$9040CC.16.16                // DAC Channel 2, X11   [Counts]
PTR DAC1X12->S.IO:$90414C.16.16                // DAC Channel 1, X12   [Counts]
PTR DAC2X12->S.IO:$9041CC.16.16                // DAC Channel 2, X12   [Counts]
```

The effective resolution of the analog output circuitry is about ~13.5 bits (±13380 software counts) spanning over the full output range of ±10V (saturates at about ~10.5 Volts). Writing to the user defined **DACnXxxInBits** pointer produces the following voltage output:

DACnXxxInBits	Single Ended [VDC]	Differential [VDC]
-13380	-10	-20
0	0	0
13380	+10	+20

Note

The output voltage is measured between AGND and DAC+ in single-ended mode. And between DAC- and DAC+ in differential mode.

➢ **SCALED DAC OUTPUT (IN VOLTS)**

The outputs can be scaled and converted into "user" voltage units. The following example PLC scales the data as needed to allow the user to command the output in units of volts:

- The global parameter(s) **DACnXxxZeroOffset** represents the voltage offset (as seen on a digital multi-meter or scope) when an output of zero is commanded. This is user adjustable.
- The global parameter **DACnXxxCtPerVolt** acts a software adjustment pot which the user can calibrate for at the rails (±10 VDC) of the output.
- The global parameter **DACnXxxVolts** is the output command in volts

Where **n** is the DAC channel number (1 or 2) of the corresponding **xx** connector (X9, X10, X11, or X12).

Example

```
GLOBAL DAC1X9Volts = 0                  // DAC Channel 1, X9    [volts]
GLOBAL DAC2X9Volts = 0                  // DAC Channel 2, X9    [volts]
GLOBAL DAC1X10Volts = 0                 // DAC Channel 1, X10   [volts]
GLOBAL DAC2X10Volts = 0                 // DAC Channel 2, X10   [volts]
GLOBAL DAC1X11Volts = 0                 // DAC Channel 1, X11   [volts]
GLOBAL DAC2X11Volts = 0                 // DAC Channel 2, X11   [volts]
GLOBAL DAC1X12Volts = 0                 // DAC Channel 1, X12   [volts]
GLOBAL DAC2X12Volts = 0                 // DAC Channel 2, X12   [volts]

GLOBAL DAC1X9ZeroOffset = 0.05          // DAC1 X9,  Zero Volt offset [volts] --USER ADJUSTABLE
GLOBAL DAC2X9ZeroOffset = 0.05          // DAC2 X9,  Zero Volt offset [volts] --USER ADJUSTABLE
GLOBAL DAC1X10ZeroOffset = 0.05         // DAC1 X10, Zero Volt offset [volts] --USER ADJUSTABLE
GLOBAL DAC2X10ZeroOffset = 0.05         // DAC2 X10, Zero Volt offset [volts] --USER ADJUSTABLE
GLOBAL DAC1X11ZeroOffset = 0.05         // DAC1 X11, Zero Volt offset [volts] --USER ADJUSTABLE
GLOBAL DAC2X11ZeroOffset = 0.05         // DAC2 X11, Zero Volt offset [volts] --USER ADJUSTABLE
GLOBAL DAC1X12ZeroOffset = 0.05         // DAC1 X12, Zero Volt offset [volts] --USER ADJUSTABLE
GLOBAL DAC2X12ZeroOffset = 0.05         // DAC2 X12, Zero Volt offset [volts] --USER ADJUSTABLE

GLOBAL DAC1X9CtPerVolt = 1338           // DAC1 X9,  Scale Factor Counts/Volt --USER ADJUSTABLE
GLOBAL DAC2X9CtPerVolt = 1338           // DAC2 X9,  Scale Factor Counts/Volt --USER ADJUSTABLE
GLOBAL DAC1X10CtPerVolt = 1338          // DAC1 X10, Scale Factor Counts/Volt --USER ADJUSTABLE
GLOBAL DAC2X10CtPerVolt = 1338          // DAC2 X10, Scale Factor Counts/Volt --USER ADJUSTABLE
GLOBAL DAC1X11CtPerVolt = 1338          // DAC1 X11, Scale Factor Counts/Volt --USER ADJUSTABLE
GLOBAL DAC2X11CtPerVolt = 1338          // DAC2 X11, Scale Factor Counts/Volt --USER ADJUSTABLE
GLOBAL DAC1X12CtPerVolt = 1338          // DAC1 X12, Scale Factor Counts/Volt --USER ADJUSTABLE
GLOBAL DAC2X12CtPerVolt = 1338          // DAC2 X12, Scale Factor Counts/Volt --USER ADJUSTABLE
```

```
OPEN PLC ExamplePLC
DAC1X9 = (DAC1X9Volts   - DAC1X9ZeroOffset)  * ABS(DAC1X9CtPerVolt)
DAC2X9 = (DAC2X9Volts   - DAC2X9ZeroOffset)  * ABS(DAC2X9CtPerVolt)
DAC1X10 = (DAC1X100Volts - DAC1X10ZeroOffset) * ABS(DAC1X10CtPerVolt)
DAC2X10 = (DAC2X100Volts - DAC2X10ZeroOffset) * ABS(DAC2X10CtPerVolt)
DAC1X11 = (DAC1X111Volts - DAC1X11ZeroOffset) * ABS(DAC1X11CtPerVolt)
DAC2X11 = (DAC2X111Volts - DAC2X11ZeroOffset) * ABS(DAC2X11CtPerVolt)
DAC1X12 = (DAC1X122Volts - DAC1X12ZeroOffset) * ABS(DAC1X12CtPerVolt)
DAC2X12 = (DAC2X122Volts - DAC2X12ZeroOffset) * ABS(DAC2X12CtPerVolt)
CLOSE
```

Note

Using this example code, the user can command the output by writing to **DACnXxxVolts** in units of volts.

Setting up the General Purpose Relay

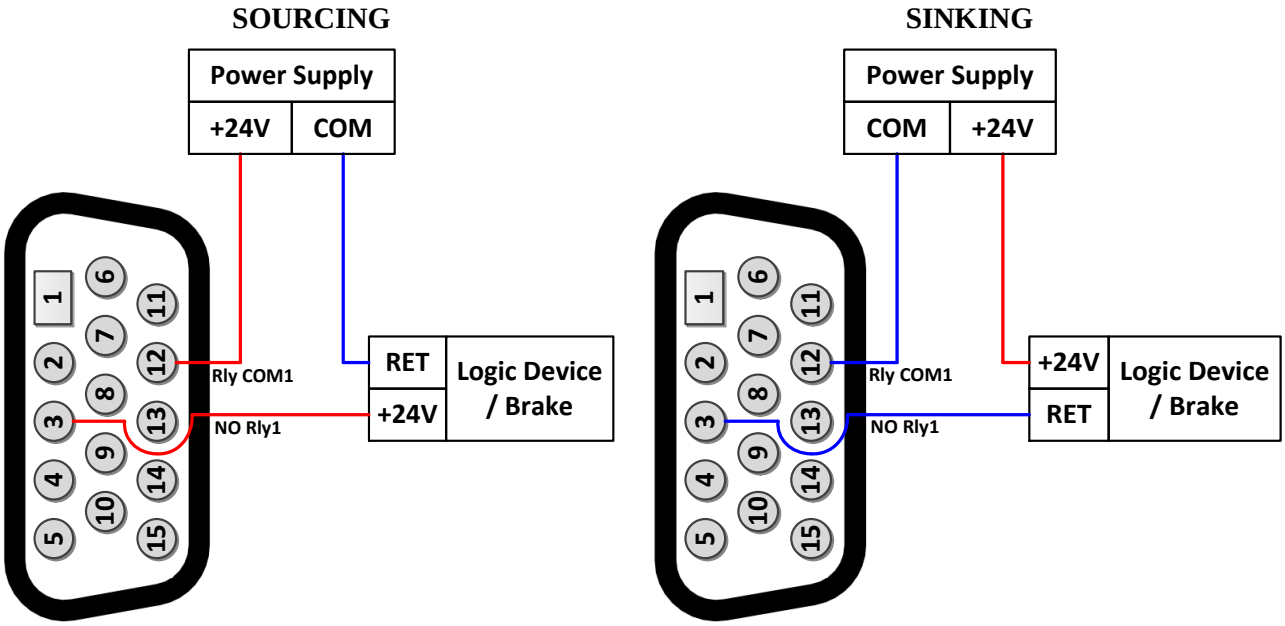
This normally open relay can be used as a general purpose relay, motor brake control, or external amplifier enable signal. It is operated by the structure element bit **PowerBrick[].Chan[].OutFlagC.**

Channel / Connector	Structure element bit	Channel / Connector	Structure element bit
Relay 1, X9	PowerBrick[0].Chan[0].OutFlagC	Relay 1, X9	PowerBrick[1].Chan[0].OutFlagC
Relay 2, X9	PowerBrick[0].Chan[1].OutFlagC	Relay 2, X9	PowerBrick[1].Chan[1].OutFlagC
Relay 1, X10	PowerBrick[0].Chan[2].OutFlagC	Relay 1, X10	PowerBrick[1].Chan[2].OutFlagC
Relay 2, X10	PowerBrick[0].Chan[3].OutFlagC	Relay 2, X10	PowerBrick[1].Chan[3].OutFlagC

If **PowerBrick[].Chan[].OutFlagC** = 0, the circuit between the common pin and the Relay pin is open.
If **PowerBrick[].Chan[].OutFlagC** = 1, the circuit between the common pin and the Relay pin is closed.

Structure Element Bit	Connection between Pin #3 and Pin #12	Connection between Pin #10 and Pin #5
PowerBrick[].Chan[].OutFlagC = 0	Open	Open
PowerBrick[].Chan[].OutFlagC = 1	Closed	Closed

The relay can be wired so that the current is either sourcing from or sinking into the Power Brick LV.



Caution

In sourcing mode, do NOT pass through voltage higher than 24VDC.

Note

The commons of the general purpose inputs / amp faults (pins #8, and #15) are tied internally to relay commons 1 and 2 respectively. If the relay is wired in sourcing mode, that general purpose input cannot be used.

The structure element bits can be assigned to user defined pointers:

```
PTR GPRelay1X9->PowerBrick[0].Chan[0].OutFlagC          // GP Relay 1, X9   =0 Open, =1 Closed
PTR GPRelay2X9->PowerBrick[0].Chan[1].OutFlagC          // GP Relay 2, X9   =0 Open, =1 Closed
PTR GPRelay1X10->PowerBrick[0].Chan[2].OutFlagC         // GP Relay 1, X10  =0 Open, =1 Closed
PTR GPRelay2X10->PowerBrick[0].Chan[3].OutFlagC         // GP Relay 2, X10  =0 Open, =1 Closed

PTR GPRelay1X11->PowerBrick[1].Chan[0].OutFlagC         // GP Relay 1, X11  =0 Open, =1 Closed
PTR GPRelay2X11->PowerBrick[1].Chan[1].OutFlagC         // GP Relay 2, X11  =0 Open, =1 Closed
PTR GPRelay1X12->PowerBrick[1].Chan[2].OutFlagC         // GP Relay 1, X12  =0 Open, =1 Closed
PTR GPRelay2X12->PowerBrick[1].Chan[3].OutFlagC         // GP Relay 2, X12  =0 Open, =1 Closed
```

If used for motor brake control (or external amplifier enable), the following settings are necessary to ensure proper synchronization with the motor channel enable/disable functions:

➢ **EXAMPLE**

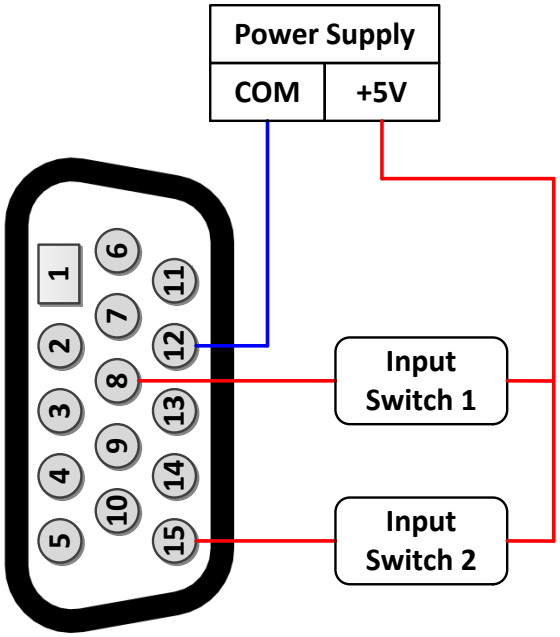
```
Motor[1].pBrakeOut = PowerBrick[0].Chan[0].OutFlagC.a   //
Motor[1].BrakeOffDelay = 5                              // msec, Brake Off Delay --USER INPUT
Motor[1].BrakeOnDelay = 5                               // msec, Brake On Delay  --USER INPUT
Motor[1].BrakeOutBit = 10                               //
```

Setting up the GP Input

This input provides a general purpose input coming from an external device (e.g. amplifier fault). It is a single-ended optically isolated input. Although 5 V is intended, a minimum voltage of only 3.5 V is required to receive the signal.

Note

The commons of the general purpose inputs / amp faults (pins #8 and #15) are tied internally to relay commons 1 and 2 respectively (pins #5 and #12). If the relay is wired in sourcing mode, creating voltage potential at the common, this GP input cannot be used.

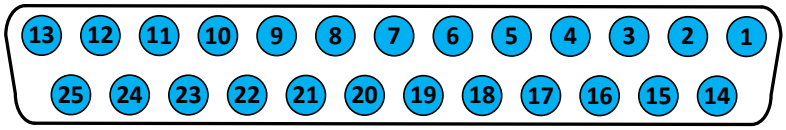


The structure element bit reflecting the status of this input is **PowerBrick[].Chan[].T**. It is a low true input, meaning it is =1 when 0 V is connected and =0 when +5 V is connected.

```
PTR GpIn1X9->PowerBrick[0].Chan[0].T          // Channel 1, X9 Input
PTR GpIn2X9->PowerBrick[0].Chan[1].T          // Channel 2, X9 Input
PTR GpIn1X10->PowerBrick[0].Chan[2].T         // Channel 1, X10 Input
PTR GpIn2X10->PowerBrick[0].Chan[3].T         // Channel 2, X10 Input

PTR GpIn1X11->PowerBrick[1].Chan[0].T         // Channel 1, X11 Input
PTR GpIn2X11->PowerBrick[1].Chan[1].T         // Channel 2, X11 Input
PTR GpIn1X12->PowerBrick[1].Chan[2].T         // Channel 1, X12 Input
PTR GpIn2X12->PowerBrick[1].Chan[3].T         // Channel 2, X12 Input
```

Limits, Flags, and EQU (X13-X14)

X13 is used to wire the limits, flags, and EQU for axes 1 – 4.
X14 is used to wire the limits, flags, and EQU for axes 5 – 8.
Per channel, there are 2 limit inputs (Plus and Minus), 2 flag inputs (Home and User), and 1 EQU output. The limits and flags are auto-regulating in the 5 – 24 VDC range. The current draw for each input is about 6 – 10 mA in the 5 – 24 VDC range. The EQU output is 5 VDC TTL level and its rise time is on the order of nanoseconds.

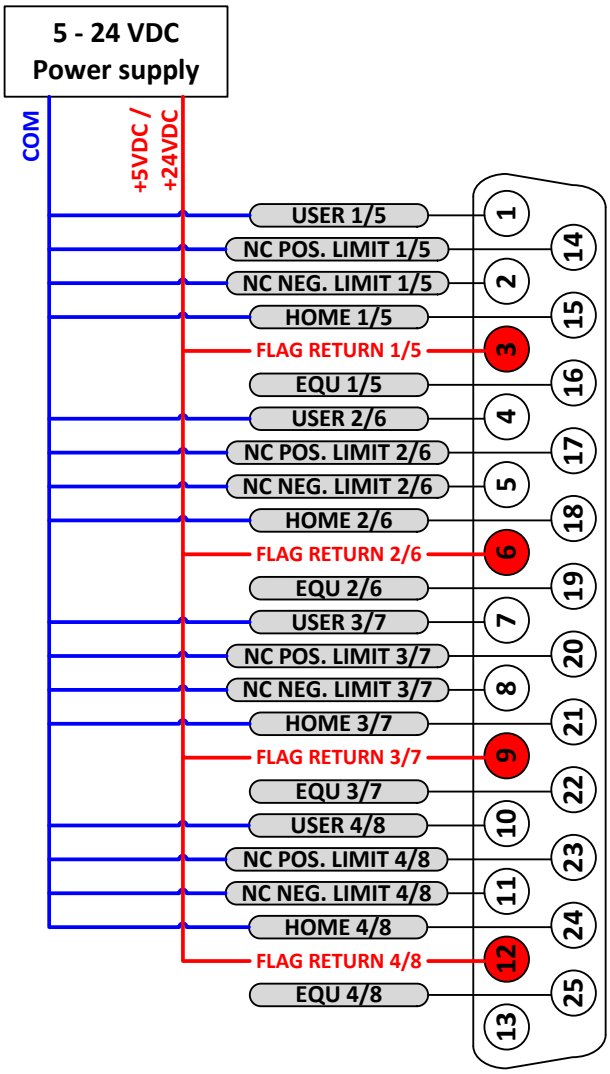
X13/X14: D-sub DB-25F **Mating: D-sub DB-25M**			
Pin #	**Symbol**	**Function**	**Description**
1	USER1/5	Input	User Flag 1/5
2	MLIM1/5	Input	Negative Limit 1/5
3	FL_RT1/5	Input	Flag Return 1/5
4	USER2/6	Input	User Flag 2/6
5	MLIM26	Input	Negative Limit 2/6
6	FL_RT2/6	Input	Flag Return 2/6
7	USER3/7	Input	User Flag 3/7
8	MLIM3/7	Input	Negative Limit 3/7
9	FL_RT3/7	Input	Flag Return 3/7
10	USER4/8	Input	User Flag 4/8
11	MLIM4/8	Input	Negative Limit 4/8
12	FL_RT4/8	Input	Flag Return 4/8
13	GND	Common	
14	PLIM1/5	Input	Positive Limit 1/5
15	HOME1/5	Input	Home Flag 1/5
16	EQU1/5	Output	Compare Output, EQU 1/5 TTL (5V) level
17	PLIM2/6	Input	Positive Limit 2/6
18	HOME2/6	Input	Home Flag 2/6
19	EQU2/6	Output	Compare Output, EQU 2/6 TTL (5V) level
20	PLIM3/7	Input	Positive Limit 3/7
21	HOME3/7	Input	Home Flag 3/7
22	EQU3/7	Output	Compare Output, EQU 3/7 TTL (5V) level
23	PLIM4/8	Input	Positive Limit 4/8
24	HOME4/8	Input	Home Flag 4/8
25	EQU4/8	Output	Compare Output, EQU 4/8 TTL (5V) level

Wiring the Limits and Flags

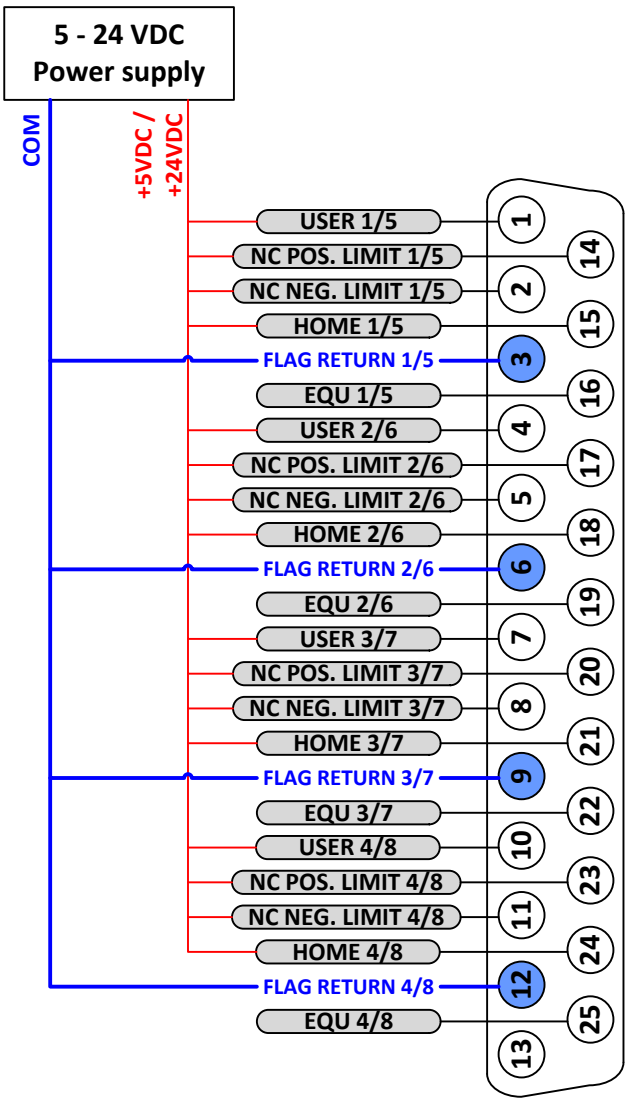
The Power Brick allows the use of sinking or sourcing limits and flags (per channel). The current flow can be from return to flag (sinking) or from flag to return (sourcing).

The overtravel limits must be normally closed switches. They can be disabled in software by setting M**otor[].pLimits** = 0 but their polarity is not software configurable.

SOURCING LIMITS AND FLAGS



SINKING LIMITS AND FLAGS



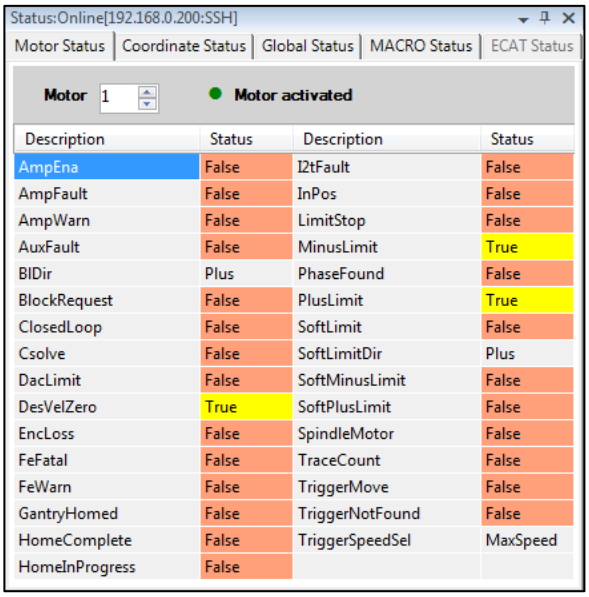
Note

The overtravel limits must be normally closed switches. They can be disabled in software, but their polarity is not configurable.

Limits and Flags Suggested Pointers

Typically, and if the corresponding channel is activated (**Motor[].ServoCtrl** = 1), the overtravel limits are monitored in the motor status window in the IDE software and the motor structure elements (i.e. **Motor[].PlusLimit**) are used in logic programming. However, if the channel is not activated, the overtravel limit inputs can be accessed through the ASIC structure elements shown below.

The User / Home flags, and EQU output are a function of the ASIC; they can always be accessed through the ASIC structure elements.



Description	Status	Description	Status
AmpEna	False	I2tFault	False
AmpFault	False	InPos	False
AmpWarn	False	LimitStop	False
AuxFault	False	MinusLimit	True
BlDir	Plus	PhaseFound	False
BlockRequest	False	PlusLimit	True
ClosedLoop	False	SoftLimit	False
Csolve	False	SoftLimitDir	Plus
DacLimit	False	SoftMinusLimit	False
DesVelZero	True	SoftPlusLimit	False
EncLoss	False	SpindleMotor	False
FeFatal	False	TraceCount	False
FeWarn	False	TriggerMove	False
GantryHomed	False	TriggerNotFound	False
HomeComplete	False	TriggerSpeedSel	MaxSpeed
HomeInProgress	False		

➢ CHANNELS 1 – 4 LIMITS AND FLAGS SUGGESTED POINTERS (X13)

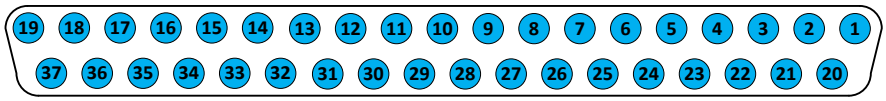
```
PTR Ch1PlusLimit->PowerBrick[0].Chan[0].PlusLimit          // Channel 1 Positive Limit
PTR Ch1MinusLimit->PowerBrick[0].Chan[0].MinusLimit        // Channel 1 Negative Limit
PTR Ch1UserFlag->PowerBrick[0].Chan[0].UserFlag            // Channel 1 User Flag
PTR Ch1HomeFlag->PowerBrick[0].Chan[0].HomeFlag            // Channel 1 Home Flag
PTR Ch1EQU->PowerBrick[0].Chan[0].Equ                      // Channel 1 EQU

PTR Ch2PlusLimit->PowerBrick[0].Chan[1].PlusLimit          // Channel 2 Positive Limit
PTR Ch2MinusLimit->PowerBrick[0].Chan[1].MinusLimit        // Channel 2 Negative Limit
PTR Ch2UserFlag->PowerBrick[0].Chan[1].UserFlag            // Channel 2 User Flag
PTR Ch2HomeFlag->PowerBrick[0].Chan[1].HomeFlag            // Channel 2 Home Flag
PTR Ch2EQU->PowerBrick[0].Chan[1].Equ                      // Channel 2 EQU

PTR Ch3PlusLimit->PowerBrick[0].Chan[2].PlusLimit          // Channel 3 Positive Limit
PTR Ch3MinusLimit->PowerBrick[0].Chan[2].MinusLimit        // Channel 3 Negative Limit
PTR Ch3UserFlag->PowerBrick[0].Chan[2].UserFlag            // Channel 3 User Flag
PTR Ch3HomeFlag->PowerBrick[0].Chan[2].HomeFlag            // Channel 3 Home Flag
PTR Ch3EQU->PowerBrick[0].Chan[2].Equ                      // Channel 3 EQU

PTR Ch4PlusLimit->PowerBrick[0].Chan[3].PlusLimit          // Channel 4 Positive Limit
PTR Ch4MinusLimit->PowerBrick[0].Chan[3].MinusLimit        // Channel 4 Negative Limit
PTR Ch4UserFlag->PowerBrick[0].Chan[3].UserFlag            // Channel 4 User Flag
PTR Ch4HomeFlag->PowerBrick[0].Chan[3].HomeFlag            // Channel 4 Home Flag
PTR Ch4EQU->PowerBrick[0].Chan[3].Equ                      // Channel 4 EQU
```

➢ CHANNELS 5 – 8 LIMITS AND FLAGS SUGGESTED POINTERS (X14)

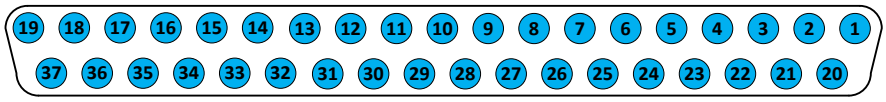
```
PTR Ch5PlusLimit->PowerBrick[1].Chan[0].PlusLimit              // Channel 5 Positive Limit
PTR Ch5MinusLimit->PowerBrick[1].Chan[0].MinusLimit            // Channel 5 Negative Limit
PTR Ch5UserFlag->PowerBrick[1].Chan[0].UserFlag                // Channel 5 User Flag
PTR Ch5HomeFlag->PowerBrick[1].Chan[0].HomeFlag                // Channel 5 Home Flag
PTR Ch5EQU->PowerBrick[1].Chan[0].Equ                          // Channel 5 EQU

PTR Ch6PlusLimit->PowerBrick[1].Chan[1].PlusLimit              // Channel 6 Positive Limit
PTR Ch6MinusLimit->PowerBrick[1].Chan[1].MinusLimit            // Channel 6 Negative Limit
PTR Ch6UserFlag->PowerBrick[1].Chan[1].UserFlag                // Channel 6 User Flag
PTR Ch6HomeFlag->PowerBrick[1].Chan[1].HomeFlag                // Channel 6 Home Flag
PTR Ch6EQU->PowerBrick[1].Chan[1].Equ                          // Channel 6 EQU

PTR Ch7PlusLimit->PowerBrick[1].Chan[2].PlusLimit              // Channel 7 Positive Limit
PTR Ch7MinusLimit->PowerBrick[1].Chan[2].MinusLimit            // Channel 7 Negative Limit
PTR Ch7UserFlag->PowerBrick[1].Chan[2].UserFlag                // Channel 7 User Flag
PTR Ch7HomeFlag->PowerBrick[1].Chan[2].HomeFlag                // Channel 7 Home Flag
PTR Ch7EQU->PowerBrick[1].Chan[2].Equ                          // Channel 7 EQU

PTR Ch8PlusLimit->PowerBrick[1].Chan[3].PlusLimit              // Channel 8 Positive Limit
PTR Ch8MinusLimit->PowerBrick[1].Chan[3].MinusLimit            // Channel 8 Negative Limit
PTR Ch8UserFlag->PowerBrick[1].Chan[3].UserFlag                // Channel 8 User Flag
PTR Ch8HomeFlag->PowerBrick[1].Chan[3].HomeFlag                // Channel 8 Home Flag
PTR Ch8EQU->PowerBrick[1].Chan[3].Equ                          // Channel 8 EQU
```

Digital I/O (X15-X16)

X15 is used to wire the general purpose digital I/Os (16 inputs and 8 outputs).

X15: D-sub DC-37F Mating: D-sub DC-37M			
Pin #	**Symbol**	**Function**	**Description**
1	GPI1	Input	Input 1
2	GPI3	Input	Input 3
3	GPI5	Input	Input 5
4	GPI7	Input	Input 7
5	GPI9	Input	Input 9
6	GPI11	Input	Input 11
7	GPI13	Input	Input 13
8	GPI15	Input	Input 15
9	IN_COM1-8	Common 01-08	Input 01 to 08 Common
10	OUT_RET	Return	Outputs Return
11	COM_EMT	Common	Outputs Common
12	GP01-	Output	Sourcing Output 1
13	GP02-	Output	Sourcing Output 2
14	GP03-	Output	Sourcing Output 3
15	GP04-	Output	Sourcing Output 4
16	GP05-	Output	Sourcing Output 5
17	GP06-	Output	Sourcing Output 6
18	GP07-	Output	Sourcing Output 7
19	GP08-	Output	Sourcing Output 8
20	GPI2	Input	Input 2
21	GPI4	Input	Input 4
22	GPI6	Input	Input 6
23	GPI8	Input	Input 8
24	GPI10	Input	Input 10
25	GPI12	Input	Input 12
26	GPI14	Input	Input 14
27	GPI16	Input	Input 16
28	IN_COM9-16	Common 09-16	Input 09 to 16 Common
29	COM_COL	Common	Outputs Common
30	GP01+	Output	Sinking Output 1
31	GP02+	Output	Sinking Output 2
32	GP03+	Output	Sinking Output 3
33	GP04+	Output	Sinking Output 4
34	GP05+	Output	Sinking Output 5
35	GP06+	Output	Sinking Output 6
36	GP07+	Output	Sinking Output 7
37	GP08+	Output	Sinking Output 8

X16 is used to wire the additional general purpose digital I/Os (16 inputs, and 8 outputs).

X16: D-sub DC-37F **Mating: D-sub DC-37M**			
Pin #	**Symbol**	**Function**	**Description**
1	GPI17	Input	Input 17
2	GPI19	Input	Input 19
3	GPI21	Input	Input 21
4	GPI23	Input	Input 23
5	GPI25	Input	Input 25
6	GPI27	Input	Input 27
7	GPI29	Input	Input 29
8	GPI31	Input	Input 31
9	IN_COM 17-24	Common 17 – 24	Input 17 to 24 Common
10	OUT_RET	Return	Outputs Return
11	COM_EMT	Common	Outputs Common
12	GPO9-	Output	Sourcing Output 9
13	GPO10-	Output	Sourcing Output 10
14	GPO11-	Output	Sourcing Output 11
15	GPO12-	Output	Sourcing Output 12
16	GPO13-	Output	Sourcing Output 13
17	GPO14-	Output	Sourcing Output 14
18	GPO15-	Output	Sourcing Output 15
19	GPO16-	Output	Sourcing Output 16
20	GPI18	Input	Input 18
21	GPI20	Input	Input 20
22	GPI22	Input	Input 22
23	GPI24	Input	Input 24
24	GPI26	Input	Input 26
25	GPI28	Input	Input 28
26	GPI30	Input	Input 30
27	GPI32	Input	Input 32
28	IN_COM_25-32	Common 25 – 32	Input 25 to 32 Common
29	COM_COL	Common	Outputs Common
30	GPO9+	Output	Sinking Output 9
31	GPO10+	Output	Sinking Output 10
32	GPO11+	Output	Sinking Output 11
33	GPO12+	Output	Sinking Output 12
34	GPO13+	Output	Sinking Output 13
35	GPO14+	Output	Sinking Output 14
36	GPO15+	Output	Sinking Output 15
37	GPO16+	Output	Sinking Output 16

About the Digital Inputs and Outputs

All general purpose inputs and outputs are optically isolated. They operate in the 12 – 24 VDC range, and can be wired to be either sinking into or sourcing out of the Power Brick.

Inputs

The inputs use the **PS2705-1NEC** photocoupler.

For sourcing inputs, connect the common lines to 12 – 24 VDC of an external power supply. The input devices are then connected to the 0V of the power supply at one end, and to the Power Brick at the other.

For sinking inputs, connect the common lines to 0V of an external power supply. The input devices are then connected to 12 – 24V of an external power supply at one end, and to the Power Brick at the other.

Note

The inputs can be wired either sourcing or sinking in sets of eight; each set possesses its own common.

Outputs

The outputs use the **PS2701-1NEC** photocoupler. They are protected with a **ZXMS6006DG**; an enhancement mode MOSFET - diode incorporated. The protection involves over-voltage, over-current, I2T and short circuit.

For sourcing outputs, connect the common lines to 12 – 24 VDC of an external power supply. The output devices are then connected to 0V of the power supply at one end, and to the Power Brick at the other.

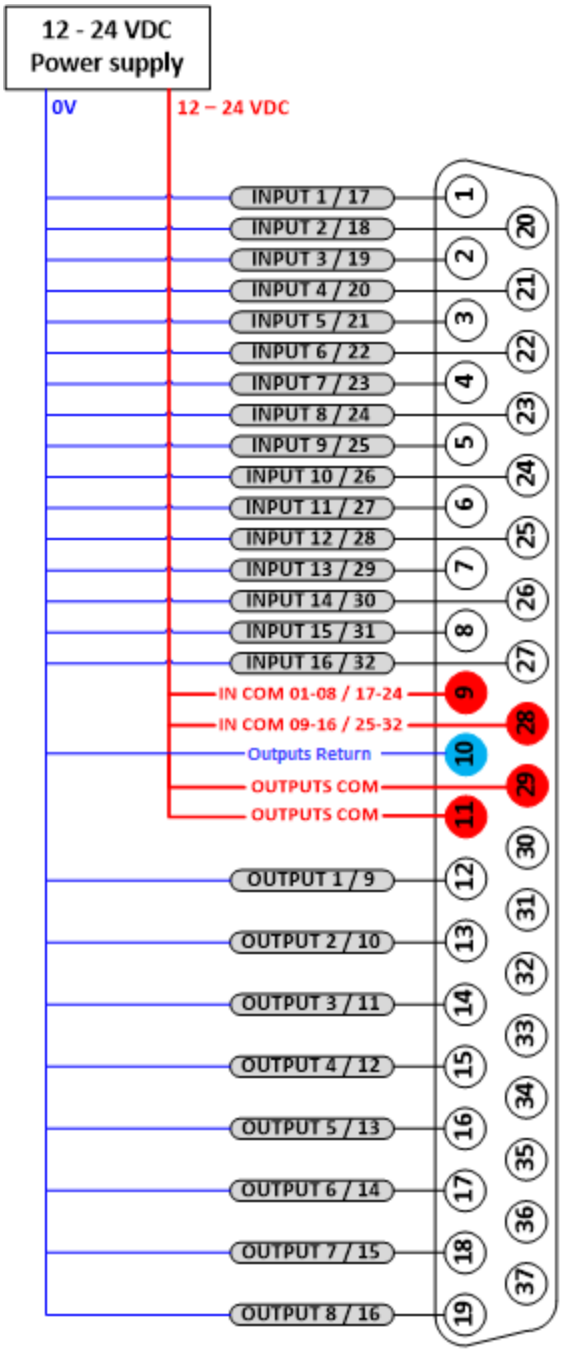
For sinking outputs, connect the common lines to 0 VDC of an external power supply. The output devices are then connected to the 12 – 24V of the power supply at one end, and to the Power Brick at the other.

Note

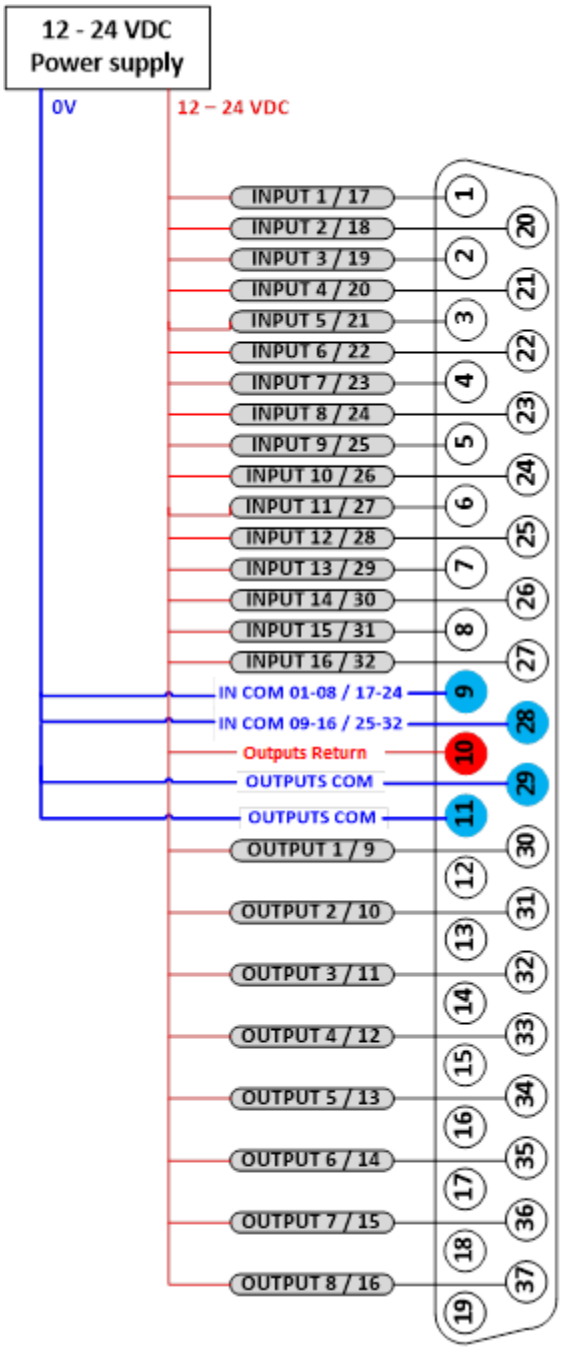
Do not mix topologies for outputs. They are all either sinking or sourcing per connector (X16 / X17).

Wiring the Digital Inputs and Outputs

➢ **SOURCING INPUTS / OUTPUTS**



➢ **SINKING INPUTS / OUTPUTS**



Note

To use outputs as PNP, wire for sourcing. To use ouputs as NPN, wire for sinking.

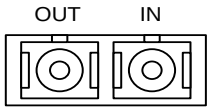
Digital I/O Pointers

```
// X15 INPUTS
PTR Input1->PowerBrick[0].GpioData[0].0.1                // Input #1, X15 Pin#1
PTR Input2->PowerBrick[0].GpioData[0].1.1                // Input #2, X15 Pin#20
PTR Input3->PowerBrick[0].GpioData[0].2.1                // Input #3, X15 Pin#2
PTR Input4->PowerBrick[0].GpioData[0].3.1                // Input #4, X15 Pin#21
PTR Input5->PowerBrick[0].GpioData[0].4.1                // Input #5, X15 Pin#3
PTR Input6->PowerBrick[0].GpioData[0].5.1                // Input #6, X15 Pin#22
PTR Input7->PowerBrick[0].GpioData[0].6.1                // Input #7, X15 Pin#4
PTR Input8->PowerBrick[0].GpioData[0].7.1                // Input #8, X15 Pin#23
PTR Input9->PowerBrick[0].GpioData[0].8.1                // Input #9, X15 Pin#5
PTR Input10->PowerBrick[0].GpioData[0].9.1               // Input #10,X15 Pin#24
PTR Input11->PowerBrick[0].GpioData[0].10.1              // Input #11,X15 Pin#6
PTR Input12->PowerBrick[0].GpioData[0].11.1              // Input #12,X15 Pin#25
PTR Input13->PowerBrick[0].GpioData[0].12.1              // Input #13,X15 Pin#7
PTR Input14->PowerBrick[0].GpioData[0].13.1              // Input #14,X15 Pin#26
PTR Input15->PowerBrick[0].GpioData[0].14.1              // Input #15,X15 Pin#8
PTR Input16->PowerBrick[0].GpioData[0].15.1              // Input #16,X15 Pin#27

// X15 OUTPUTS                                                    Sourcing   Sinking
PTR Output1->PowerBrick[0].GpioData[0].16.1              // Output #1,X15  Pin#12     Pin#30
PTR Output2->PowerBrick[0].GpioData[0].17.1              // Output #2,X15  Pin#13     Pin#31
PTR Output3->PowerBrick[0].GpioData[0].18.1              // Output #3,X15  Pin#14     Pin#32
PTR Output4->PowerBrick[0].GpioData[0].19.1              // Output #4,X15  Pin#15     Pin#33
PTR Output5->PowerBrick[0].GpioData[0].20.1              // Output #5,X15  Pin#16     Pin#34
PTR Output6->PowerBrick[0].GpioData[0].21.1              // Output #6,X15  Pin#17     Pin#35
PTR Output7->PowerBrick[0].GpioData[0].22.1              // Output #7,X15  Pin#18     Pin#36
PTR Output8->PowerBrick[0].GpioData[0].23.1              // Output #8,X15  Pin#19     Pin#37
```

```
// X16 INPUTS
PTR Input17->PowerBrick[1].GpioData[0].0.1               // Input #17,X16 Pin#1
PTR Input18->PowerBrick[1].GpioData[0].1.1               // Input #18,X16 Pin#20
PTR Input19->PowerBrick[1].GpioData[0].2.1               // Input #19,X16 Pin#2
PTR Input20->PowerBrick[1].GpioData[0].3.1               // Input #20,X16 Pin#21
PTR Input21->PowerBrick[1].GpioData[0].4.1               // Input #21,X16 Pin#3
PTR Input22->PowerBrick[1].GpioData[0].5.1               // Input #22,X16 Pin#22
PTR Input23->PowerBrick[1].GpioData[0].6.1               // Input #23,X16 Pin#4
PTR Input24->PowerBrick[1].GpioData[0].7.1               // Input #24,X16 Pin#23
PTR Input25->PowerBrick[1].GpioData[0].8.1               // Input #25,X16 Pin#5
PTR Input26->PowerBrick[1].GpioData[0].9.1               // Input #26,X16 Pin#24
PTR Input27->PowerBrick[1].GpioData[0].10.1              // Input #27,X16 Pin#6
PTR Input28->PowerBrick[1].GpioData[0].11.1              // Input #28,X16 Pin#25
PTR Input29->PowerBrick[1].GpioData[0].12.1              // Input #29,X16 Pin#7
PTR Input30->PowerBrick[1].GpioData[0].13.1              // Input #30,X16 Pin#26
PTR Input31->PowerBrick[1].GpioData[0].14.1              // Input #31,X16 Pin#8
PTR Input32->PowerBrick[1].GpioData[0].15.1              // Input #32,X16 Pin#27

// X16 OUTPUTS                                                   Sourcing   Sinking
PTR Output9->PowerBrick[1].GpioData[0].16.1              // Output #9, X16  Pin#12     Pin#30
PTR Output10->PowerBrick[1].GpioData[0].17.1             // Output #10,X16  Pin#13     Pin#31
PTR Output11->PowerBrick[1].GpioData[0].18.1             // Output #11,X16  Pin#14     Pin#32
PTR Output12->PowerBrick[1].GpioData[0].19.1             // Output #12,X16  Pin#15     Pin#33
PTR Output13->PowerBrick[1].GpioData[0].20.1             // Output #13,X16  Pin#16     Pin#34
PTR Output14->PowerBrick[1].GpioData[0].21.1             // Output #14,X16  Pin#17     Pin#35
PTR Output15->PowerBrick[1].GpioData[0].22.1             // Output #15,X16  Pin#18     Pin#36
PTR Output16->PowerBrick[1].GpioData[0].23.1             // Output #16,X16  Pin#19     Pin#37
```

MACRO (X17)

If a MACRO option was selected, the Power Brick LV provides the following connector for MACRO communications:

MACRO SC-Style Fiber Connector		
Pin #	**Symbol**	**Function**
1	IN	MACRO Ring Receiver
2	OUT	MACRO Ring Transmitter

Note

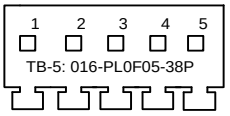
The fiber optic version of MACRO uses 62.5/125 multi-mode glass fiber optic cable terminated in an SC-style connector. The optical wavelength is 1,300 nm.

The input connector must be inserted into the MACRO output connector of the previous device on the MACRO ring. The output connector must be inserted into the input MACRO connector of the next device on the MACRO ring.

Abort and Watchdog (X18)

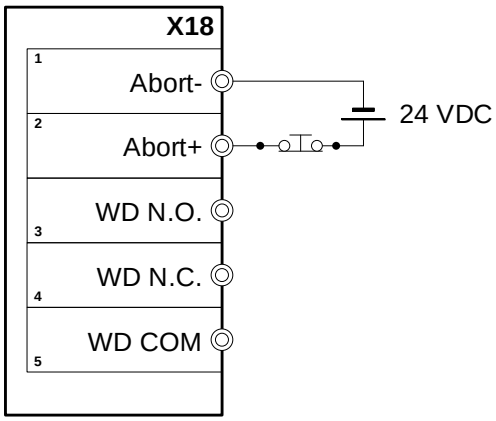
X18 has two essential functions:

- Global abort input.
- Watchdog output.

X15: Phoenix 5-pin TB Female Mating: Phoenix 5-pin TB Male			
Pin #	**Symbol**	**Function**	**Notes**
1	ABORT-	Input	ABORT Return
2	ABORT+	Input	ABORT Input 24VDC
3	WD N.O.	Output	Watchdog (normally open contact)
4	WD N.C.	Output	Watchdog (normally closed contact)
5	WD COM	Common	Watchdog common
Phoenix Contact Mating Connector Part #1850699			

Abort Input

The 24 VDC abort input provides a "category 2" controlled safe stop under the IEC-61800-5-2 machine safety standard, suitable for applications such as aborting motion for opening machine door, or replacing tool(s). If an Abort input button is used, it must be a normally closed switch:



If a "Category 1" safe stop under this standard – a controlled stop followed by a software-free disabling – is desired, the same action that toggles this input should also start a qualified time-delay relay which will then drop out power from a key circuit; usually either bus power (E-Stop circuit) or gate-driver power (STO).

Note

This "global abort" function is not suitable by itself in cases where power must be removed from the motor (such as Category 0 or 1 under the IEC-61800-5-2 machine safety standard) is required.

➢ **GLOBAL ABORT KEY SETTINGS**

The global abort input is enabled / disabled in software through **Sys.pAbortAll**.

Sys.pAbortAll	= **PowerBrick[0].GpioData[0].a**	Global abort input enabled
	= 0	Global abort input disabled

```
Sys.pAbortAll = PowerBrick[0].GpioData[0].a          // =0 global abort disabled
Sys.AbortAllBit = 31                                 // Default, do not change
Sys.AbortAllLimit = 5                                // Scan count limit threshold
```

The status of the global abort input can be monitored in the global status window in the IDE software, also it can be queried using the system element **Sys.AbortAll** (=0 deactivated or armed, =1 triggered).
The structure element **Coord[].AbortAllMode** specifies how a group of motors in a coordinate system behaves in the event of an abort trigger:

Coord[].AbortAllMode	Global Abort Trigger Action	Equivalent Software Command
= 0	All motors in the coordinate system are brought to a closed-loop controlled stop.	Abort
= 1	All motors in the coordinate system are immediately disabled (killed) without delay for brake engagement.	Disable
= 2	All motors in the coordinate system are first brought to a closed-loop controlled stop then as each motor reaches a desired velocity of zero, it executes a “delayed kill”, with an immediate engagement of the brake (specified by **Motor[].pBrakeOut**) followed by a disabling of the motor after the interval specified by **Motor[].BrakeOnDelay**.	Adisable
= 3	The coordinate system is not affected by the global abort input.	-

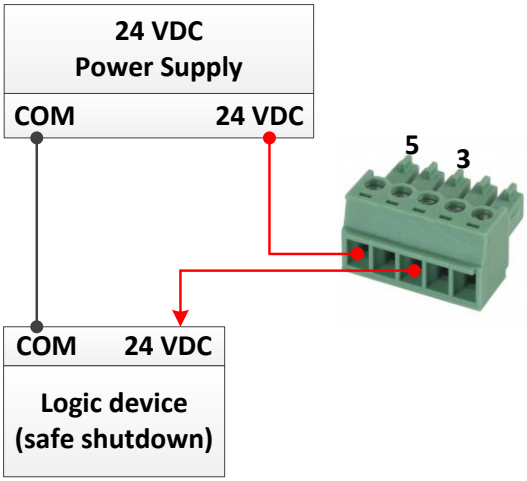
Note

By default, all motors are assigned to coordinate system 0.

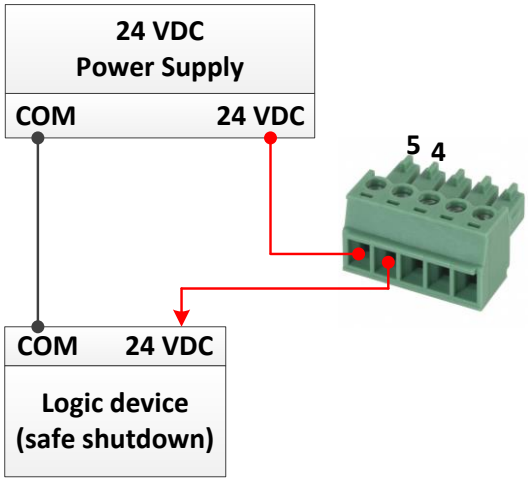
Watchdog Relay

The Watchdog relay(s) allows the user to connect to safety circuit in order to bring the machine to a stop in a safe manner in the occurrence of a watchdog. Normally open or closed contacts are available:

NORMALLY OPEN



NORMALLY CLOSED



The watchdog relay(s) are triggered when:

- A hard watchdog occurs, interrupting communication and killing all tasks.
- A soft watchdog occurs, killing all tasks (**Sys.WDTFault** = 1 or 2). Communication remains alive in this case.

Operation	Mode	Connection between pins #5 and #3	Connection between pins #5 and #4
Watchdog	Not Triggered	Closed	Open
	Triggered	Open	Closed

External Encoder Power Supply (X19)

Typically, feedback devices power is supplied through the X1 – X8 connectors using the internal +5VDC power supply. However, if the total feedback devices power budget exceeds ~ 2 amperes, this connector can be used to bring in the power supply from an external source.

Caution

Encoders requiring voltage supply levels other than +5 VDC must be supplied externally, neither through X1 – X8 nor through X19.

Caution

The maximum current draw out of a single encoder channel must not exceed 500 mA.

Wiring the Encoder Supply

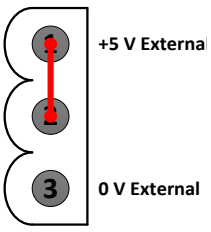
Pin#	Symbol	Description	Note
1	5 VDC External	+5 VDC Input when using external supply	
2	–	+5 VDC Output	Tie to pin#1 to use internal power supply
3	GND	0 VDC Input when using external supply	
Mating Connector Phoenix Contact P/N: 1778845			

Caution

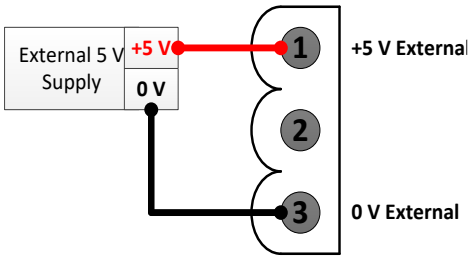
Only two of the three available pins should be used at one time. Do not daisy-chain the internal 5V power supply with an external one.

By default, pins 1 – 2 are tied together to use the internal power supply. To wire an external power supply, remove the jumper tying pins 1 – 2 and connect the external +5 V to pin #1, and 0 V to pin #3:

Internal Power Supply Wiring (Default)



External Power Supply Wiring



Note

A jumper tying pins 1 and 2 is the default configuration. This is the configuration in which the Power Brick LV is shipped.

Note

The controller's (PMAC's) 5 VDC logic is independent of this scheme, so if no encoder power is provided the PMAC will remain powered-up (provided the standard 24 volts is brought in).

Functionality and Safety Considerations

There are a couple of safety and functionality measures to take into account when an external encoder power supply is utilized:

- Power sequence: encoders versus controller/drive
  It is highly recommended to power up the encoders before applying power to the Power Brick LV
- Encoder Power Loss (i.e. power supply failure, loose wire/connector)

The Power Brick LV, with certain feedback devices, can be set up to read absolute position or perform phasing on power-up (either automatic firmware functions, or user written PLCs). If the encoder power is not available, these functions will not be performed properly. Trying to close the loop on a motor without encoder feedback could be dangerous.

Caution

Make sure that the encoders are powered-up before executing any motor/motion commands.

Losing encoder power can lead to dangerous runaway conditions; setting up encoder loss protection, fatal following error limits, and I2T protection are highly advised.

Caution

Make sure that the encoder loss protection is active, fatal following error limit is set tightly, and I2T is configured.

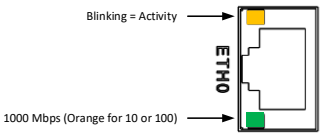
With commutated motors (i.e. DC brushless), a loss of encoder generally breaks the commutation cycle causing a fatal following error or I2T fault either in PMAC or amplifier side. However, with non-commutated motors (i.e. DC brush), losing encoder signal can more likely cause dangerous runway conditions.

RTETH & Fieldbus (X20-X23)

Refer to the ACC-72EX Manual for this connector's pinout and functionality.

ETH0 and ETH1/ECAT

The Power Brick LV comes with two RJ-45 ports on the front panel: ETH 0 and ETH1/ECAT. A Category 5e network cable should be used for these connections. Both ports provide transformer isolation to prevent ground-loop problems.

ETH0 and ETH1/ECAT: 8-Pin RJ45 Receptacle			
Pin #	**Symbol**	**Function**	**Description**
1	P0MDI0+	BIDIR	LINE 0 POS
2	P0MDI0-	BIDIR	LINE 0 NEG
3	P0MDI1+	BIDIR	LINE 1 POS
4	P0MDI1-	BIDIR	LINE 1 NEG
5	P0MDI2+	BIDIR	LINE 2 POS
6	P0MDI2-	BIDIR	LINE 2 NEG
7	P0MDI3+	BIDIR	LINE 3 POS
8	P0MDI3-	BIDIR	LINE 3 NEG

A blinking amber light on the top side of the connector indicates activity. Ethernet speed will be auto-negotiated and prefer 1000 Mbps, which is indicated by a solid green light on the bottom side of the connector. A solid orange light instead indicates a 10 or 100 Mbps connection.

ETH0 Ethernet Port

The ETH 0 port is the bottom Ethernet connector on the front panel. It is the primary port for communicating with the CPU board from a host computer, as when using the Integrated Development Environment (IDE) program running on a Windows™ PC for developing your application.

Note

Multiple computers on a single network can independently communicate to the Power PMAC board through this single hardware port.

ETH1/ECAT Port

The ETH1/ECAT port is the second-to-bottom-connector on the front panel. If the option for no EtherCAT was purchased, it is the auxiliary Ethernet port and not intended for primary host communications, but can be used to communicate to peripheral devices. If any option for EtherCAT was purchased, it can be used to connect to EtherCAT devices in a line or star topology.

USB and Diagnostic

The Power Brick LV provides two USB ports on the front panel, one host port labled “USB“ and one serial/device port labeled “DIAG.”. Both provide USB 2.0 protocol communications.

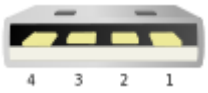
Caution

USB ports are not electrically isolated, so care must be taken in the grounding scheme when any separately powered device is connected to one of these ports. Poor-quality communications and even permanent component damage is possible when ground loop issues or significant differences in ground potential exist.

USB Host Port

The USB “host” port is labeled “USB 1” on the front panel. It is a “Standard-A” format connector located just above the Ethernet ports and has a horizontal orientation. With this port, the Power PMAC CPU acts as the host computer and various peripheral devices can be connected through this port. This connector should be used for Host PC to Power PMAC communication.

Probably the most common peripheral device used on this port is the “USB stick” flash drive. The Power PMAC CPU board will automatically recognize standardly formatted flash drives connected to this port. It is even possible to boot the CPU from this drive if the proper boot files are present on the drive. It is also possible to use USB peripheral devices such as true disk drives and keyboards.

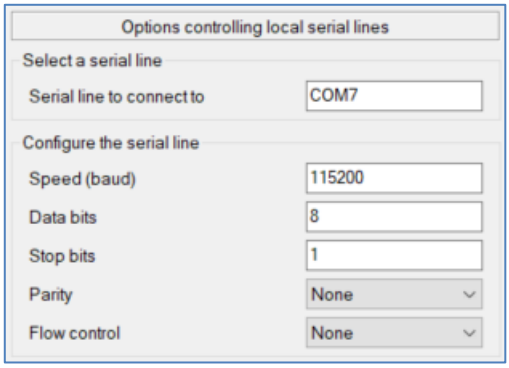
USB 1: 4-Pin Receptacle				
Pin #	**Symbol**	**Function**	**Description**	**Notes**
1	VCC	OUTPUT	SUPPLY VOLTAGE	
2	D-	BIDIRECT.	DATA NEG.	
3	D+	BIDIRECT.	DATA POS.	
4	GND	COMMON	REF. VOLTAGE	

This connector provides a USB “host” interface on a Standard A connector. It is suitable for standard USB connectors to external devices.

USB-Serial UART Diagnostic Port

The second USB port is labeled “DIAG.” on the front panel. It is a “Micro-B” format connector located just above the USB host port. The function of this port in controlled by the “DIAG. MODE SELECT” LED (top) and button (bottom) just to the right of the USB ports.

When the port is in its default position, indicated by a green light, it acts as a serial communications port. The following settings can be used to connect through PuTTY once the COM number is found in the device manger.

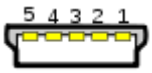


Note

This is a debug terminal that can be used for communicating with Power PMAC in the event that Ethernet communication fails (e.g. to acquire an IP address when unknown). In general, this port should not be used to communicate to external peripherals, but rather left in case of the need to debug.

Press the button with a bent paperclip or small screwdriver to put the port into mass storage mode, indicated by a yellow light. In this mode, the Power PMAC CPU board acts as a peripheral device when it is powered off. That is, you can access Power PMAC’s flash memory with a host computer by first powering down Power PMAC, connecting it to the host device through this USB port and pressing the diagnostic mode select button. Power PMAC will then act just like a USB flash drive. This is useful for device imaging and for recovering Power PMAC projects which were stored in flash memory in the event that the Power PMAC is somehow damaged or stops functioning.

Its pinout is below:

USB 2: 5-Pin Receptacle				
Pin #	**Symbol**	**Function**	**Description**	**Notes**
1	VCC	OUTPUT	SUPPLY VOLTAGE	
2	D-	BIDIRECT.	DATA NEG.	
3	D+	BIDIRECT.	DATA POS.	
4	ID	OUTPUT	BUS TYPE IDENT	
5	GND	COMMON	REF. VOLTAGE	

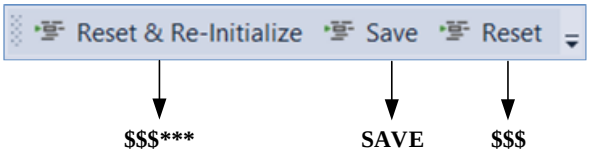
MANUAL MOTOR CONFIGURATION

This section describes the step-by-step procedure for setting up motors with the Power Brick LV.

Step 1: Creating an IDE Project

Reset

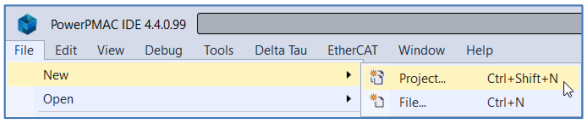
For new projects, starting from factory default settings is highly recommended to ensure a clean starting point. This is performed by issuing a global (factory default) reset **$$$***,** followed by a **SAVE,** and a normal reset **$$$**. The IDE toolbar offers shortcuts to these commands as an alternative to typing them in the terminal window.

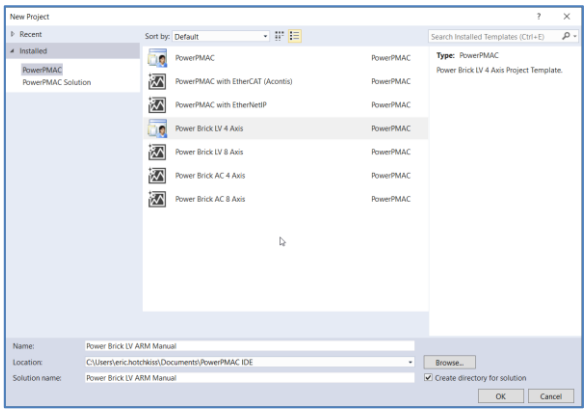


New Project

To create a new project:

- File
- New
- Project
- Choose Power Brick LV Template based (per model type)
- Give the project a name and folder location >> OK

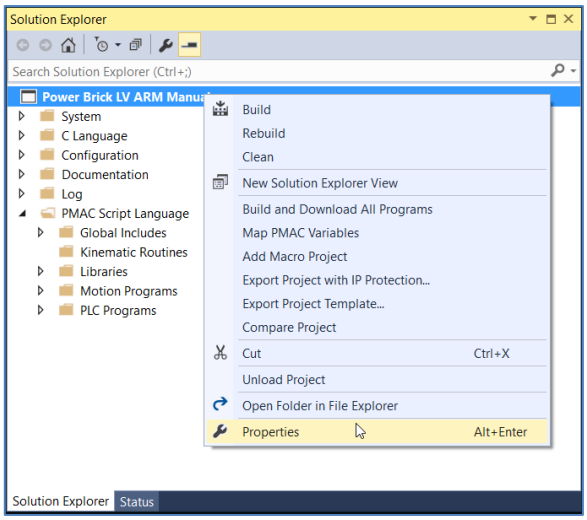


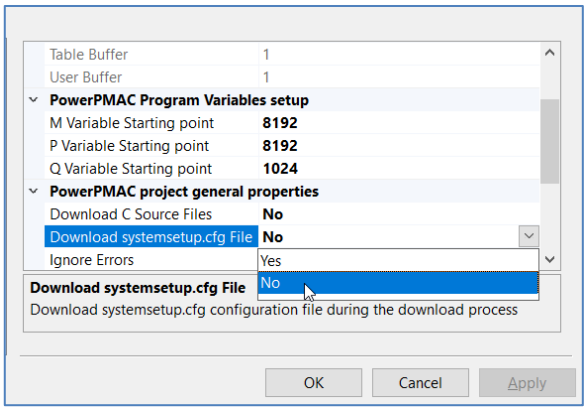


Disable Systemsetup Download

Setting up motors manually does not require using the System Setup tool. The configuration resulting from the System Setup tool must be disabled in this method.

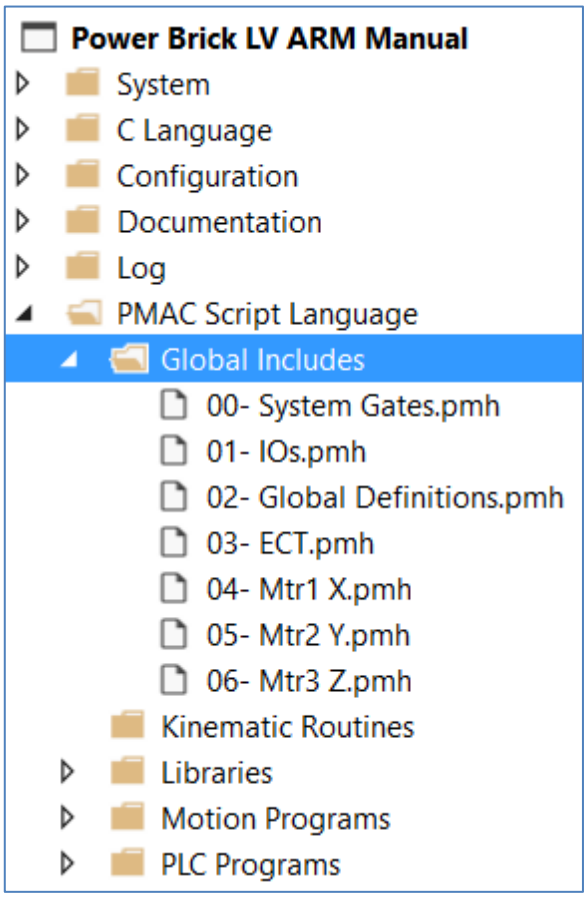
- Right-click on project name
- Properties
- Download Systemsetup.cfg File >> NO >> OK





Recommended Project Layout

The majority of the parameters described in the following sections are typically placed under Global Includes. Files in this folder can be managed per the user preference. They can be added, deleted, inserted from an existing project, re-named, organized (moved up and down) etc… Refer to the IDE Manual to learn about these manipulations. One recommended layout is as follows:



File	Typical Content	Example
System Gates	System parameters	Sys.MaxMotors
	Gate parameters	Gate3[0].PhaseFreq
	Channel parameters	Gate3[0].Chan[0].PwmFreqMult
	Power Brick LV specific	BrickLV.TwoPhaseMode
IOs	Digital I/O pointers	PTR Input1->PowerBrick[0].GpioData[0].0.1
	Analog I/O pointers	PTR ADC1X9->S.IO:$900028.16.16
Global Definitions	User-defined variables	GLOBAL MyVar
ECT	Encoder Conversion Table	EncTable[1].Type
Motor (e.g. Mtr1 X)	Motor parameters	Motor[1].ServoCtrl

Step 2: Basic Optimization and System Gates Settings

The parameters in this sections are typically placed in the System Gates.pmh file.

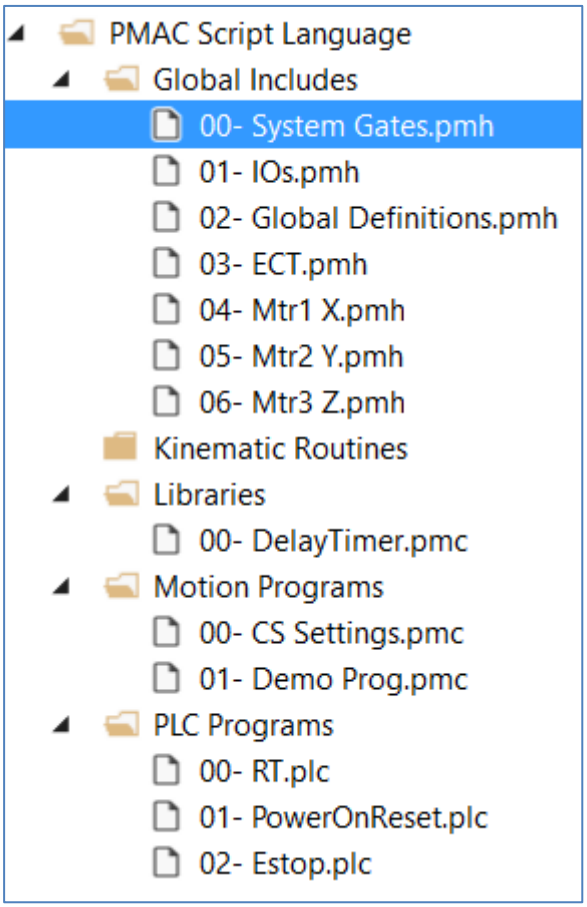
Write Protect Key, Sys.WpKey

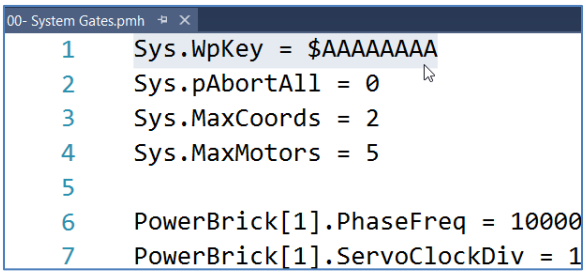
Many DSPGATE **Gate3[]** or the alias **PowerBrick[]** structures are write-protected by the firmware. They cannot be changed unless **Sys.WpKey** is set to the proper value. To disable write-protection:

```
Sys.WpKey = $AAAAAAAA
```

It is best to place this parameter setting in the beginning of the System Gates.pmh file. This will affect all subsequent script files, and reduces the risk of forgetting to set it up in subsequent sections.

Example





Note

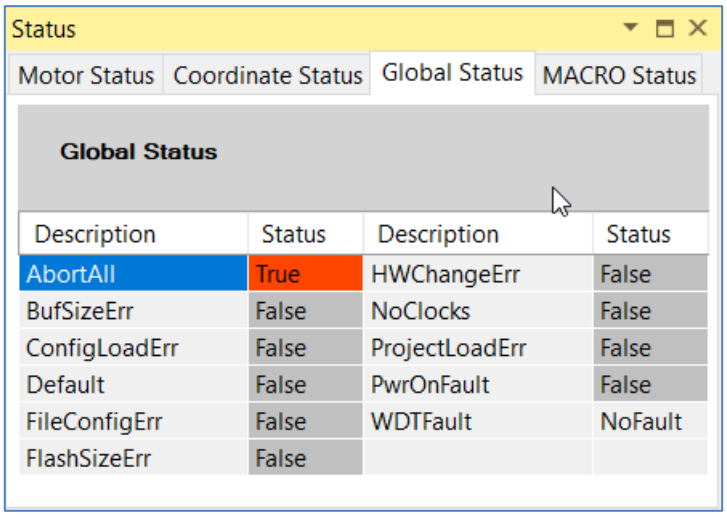
Changing write-protected structures with the write-protection ENABLED does NOT produce an error. It is the user's responsibility to make sure that **Sys.WpKey** is set up accordingly.

Abort All Input, Sys.pAbortAll

If the abort input X18 is not wired or disabled in software (**Sys.pAbortAll = 0**), PMAC will try to close the loop on the motor every time it is enabled which could cause the motor the move or jump if it has not been set up correctly yet.

The abort input X18 must be wired or disabled in software (**Sys.pAbortAll = 0**) prior to attempting to set up or enable a motor.

If the +24 VDC abort input is not wired or disabled in software (**Sys.pAbortAll = 0**), PMAC will try to close the loop on the motor every time it is enabled. This could prevent setting up a motor properly, such as phasing manually or performing an open loop test. If the abort input is triggered, the global status bit **AbortAll** will be true.



Maximum Number of Motors, Sys.MaxMotors

One of the basic and important optimization parameters for CPU load management is **Sys.MaxMotors** which specifies the highest number of motor (plus 1 including the built-in Motor #0).

Examples

- For setting up motors 1 through 4, Sys.MaxMotors should be set to 5.
- For setting up motors 1 through 9, Sys.MaxMotors should be set to 10.

Maximum Number of Coordinate Systems, Sys.MaxCoords

One of the basic and important optimization parameters for CPU load management is **Sys.MaxCoords** which specifies the highest number of Coordinate System CS (plus 1 including the built-in CS 0).

Examples

- For setting up two coordinate systems 1 and 2, Sys.MaxCoords should be set to 3.
- For setting up 4 coordinates systems 1, 2, 3, and 4, Sys.MaxCoords should be set to 5.

Dominant Clock Frequencies

The choice of clock frequencies relies typically on the system requirements, hardware, and type of application.

Phase:	The phase clock governs the current loop calculation, current sensor readings, and user written phase routine. Typically, the maximum phase clock frequency should not exceed twice that of the PWM. Setting it faster is meaningless and will not result in any performance enhancement.
PWM:	The PWM clock governs the command output to the amplifier. In motor applications, it is directly related to the inductance and resistance of the motor. A minimum value can be determined empirically as shown in the equation below.
Servo:	The Servo clock governs primarily the servo process (encoder read, motor command), and user written servo routine(s). Higher servo frequencies result, in general, in improved performance. The need for increasing the servo clock could come from several factors such as high speed/precision applications, synchronizing to external events, high speed position capture/compare, and kinematics calculation. High resolution encoders (e.g. serial, sinusoidal), linear motors, and galvanometers are usually set up with higher servo rates for best results.
EtherCAT:	The EtherCAT clock governs the rate at which data is transferred with slave devices. For EtherCAT drives, this limits servo performance. It is usually set the same as the servo clock.
Hardware:	The hardware clocks govern the sampling rate of encoders, digital /analog converters, and control the pulse frequency modulation PFM output.

Minimum PWM Frequency

The minimum PWM frequency for a motor application can be computed empirically using the time constant (τ) of the motor which is calculated dividing the motor inductance (L) by the resistance (R). Both values are phase to phase. The minimum PWM Frequency (PWM_{min}) is then determined using the following relationship:

$\tau\ (sec) = \frac{L\ (H)}{R\ (Ohms)}$	$PWM_{min}\ (Hz) = \frac{20}{2\pi * \tau\ (sec)} = \frac{10 * R\ (Ohms)}{\pi * L\ (H)}$

Example: A motor with an inductance of 2.80 mH and a resistance of 14 Ω (phase-phase) yields a time constant of 200 μsec. Therefore, the minimum PWM Frequency should be about ~16 kHz.

Note

For many motors the Minimum PWM Frequency is low enough not to matter In this case, the nominal PWM frequency should be used.

Note

The nominal PWM frequency is 40 KHz for 0.25/0.75A, and 1/3A channels, and 20 KHz for 5/15A channels.

Recommended Clock Frequencies

The recommended clock frequency settings for the Power Brick LV are 20 kHz PWM – as a minimum, or higher depending on the power rating of the channel – 10 kHz Phase, and 5 kHz Servo.

- **Sys.ServoPeriod** and **Sys.PhaseOverServoPeriod** are critical for proper implementation of the clock settings. Make sure equations are computed.
- **Sys.RtIntPeriod** specifies the cycle of the "real-time interrupt".
- The Servo frequency is determined from the phase clock using the following equation:

$$f_{servo} = \frac{f_{Phase}}{\text{PowerBrick[].Chan[].ServoClockDiv} + 1}$$

- The PWM frequency is determined from the phase clock using the following equation:

$$f_{PWM} = \frac{\text{PowerBrick[].Chan[].PwmFreqMult} + 1}{2} \times f_{Phase}$$

Note

A **Save**, followed by a **$$$** or power cycle is strongly advised after changing clock settings.

Note

Clock setting parameters require DISABLING write-protection **Sys.WpKey = $AAAAAAAA** in order to take effect.

Data Unpacking

The ADC inputs and motor phase outputs' data is packed by default in the Power PMAC firmware into single 32-bit registers. Typically, this improves the efficiency of the computation algorithms, especially in extremely high performance applications or with a large number of axes (up to 256).

However, this enhancement may not be as noteworthy with the Power Brick LV considering the significantly lower number of axes it is usually controlling. Also, the Power Brick LV offers many functions that do not support packed data which mandates unpacking them:

Note

Unpacking the IN and OUT data is critical for the proper operation of the Power Brick LV.

Setting up the BrickLV Structure Elements

The **BrickLV** data structure elements are setup and status parameters pertaining to the Power Brick LV firmware. They allow direct communication with the amplifier processor.

The **BrickLV** data structure elements consist of global (affecting all motor channels) and channel specific parameters. Certain elements can be saved others are read-only, volatile, or self-resetting.

The complete list and description of the **BrickLV** data structure elements can be found in the BrickLV Structure Elements section of this manual.

Starting from factory default settings, the necessary and sufficient **BrickLV** elements for setting up a motor safely and properly are:

BrickLV.Chan[].TwoPhaseMode	= 0 = 1	for brushless / brush motors (default) For stepper motors, direct micro-stepping
BrickLV.Chan[].I2tWarnOnly	= 0 = 1	Kill motor, display fault (default) Don't kill motor, report warning to the status register
BrickLV.Reset	= 1	To clear faults and save TwoPhaseMode, and I2tWarnOnly settings. Must wait for fail/pass confirmation of the operation.

Caution

As shown in the Power-On Reset PLC example, it is strongly recommended for users to confirm the pass/fail status of the reset (**BrickLV.Reset = 1**) process.

Caution

Querying the value of **BrickLV.Chan[].TwoPhaseMode** does NOT guarantee that the returned value is what the amplifier channel output mode is set to. **BrickLV.Reset = 1** must have executed at least once successfully for the **TwoPhaseMode** setting to be applied and saved.

Caution

BrickLV.Reset should NOT be saved **= 1**, but rather set in the power-on reset plc.

Note

The **TwoPhaseMode** and **I2TWarnOnly** elements can be saved into the active memory.

System Gates Sample File for PBL4 Brushless/Brush

```
Sys.WpKey = $AAAAAAAA
Sys.pAbortAll = 0
Sys.MaxCoords = 2
Sys.MaxMotors = 5
PowerBrick[0].PhaseFreq = 10000
PowerBrick[0].ServoClockDiv = 1
Sys.RtIntPeriod = 0
Sys.ServoPeriod = 1000 * (PowerBrick[0].ServoClockDiv + 1) / PowerBrick[0].PhaseFreq
Sys.PhaseOverServoPeriod = 1 / (PowerBrick[0].ServoClockDiv + 1)

PowerBrick[0].Chan[0].PwmFreqMult = 3
PowerBrick[0].Chan[1].PwmFreqMult = 3
PowerBrick[0].Chan[2].PwmFreqMult = 3
PowerBrick[0].Chan[3].PwmFreqMult = 3

PowerBrick[0].Chan[0].PackOutData = 0
PowerBrick[0].Chan[1].PackOutData = 0
PowerBrick[0].Chan[2].PackOutData = 0
PowerBrick[0].Chan[3].PackOutData = 0

PowerBrick[0].Chan[0].PackInData = 0
PowerBrick[0].Chan[1].PackInData = 0
PowerBrick[0].Chan[2].PackInData = 0
PowerBrick[0].Chan[3].PackInData = 0
```

System Gates Sample File for PBL4 Stepper

```
Sys.WpKey = $AAAAAAAA
Sys.pAbortAll = 0
Sys.MaxCoords = 2
Sys.MaxMotors = 5
PowerBrick[0].PhaseFreq = 10000
PowerBrick[0].ServoClockDiv = 1
Sys.RtIntPeriod = 0
Sys.ServoPeriod = 1000 * (PowerBrick[0].ServoClockDiv + 1) / PowerBrick[0].PhaseFreq
Sys.PhaseOverServoPeriod = 1 / (PowerBrick[0].ServoClockDiv + 1)

PowerBrick[0].Chan[0].PwmFreqMult = 3
PowerBrick[0].Chan[1].PwmFreqMult = 3
PowerBrick[0].Chan[2].PwmFreqMult = 3
PowerBrick[0].Chan[3].PwmFreqMult = 3

PowerBrick[0].Chan[0].PackOutData = 0
PowerBrick[0].Chan[1].PackOutData = 0
PowerBrick[0].Chan[2].PackOutData = 0
PowerBrick[0].Chan[3].PackOutData = 0

PowerBrick[0].Chan[0].PackInData = 0
PowerBrick[0].Chan[1].PackInData = 0
PowerBrick[0].Chan[2].PackInData = 0
PowerBrick[0].Chan[3].PackInData = 0

BrickLV.Chan[0].TwoPhaseMode = 1
BrickLV.Chan[1].TwoPhaseMode = 1
BrickLV.Chan[2].TwoPhaseMode = 1
BrickLV.Chan[3].TwoPhaseMode = 1
```

System Gates Sample File for PBL8 Brushless/Brush

```
Sys.WpKey = $AAAAAAAA
Sys.pAbortAll = 0
Sys.MaxCoords = 2
Sys.MaxMotors = 9

PowerBrick[1].PhaseFreq = 10000
PowerBrick[1].ServoClockDiv = 1

PowerBrick[0].PhaseFreq = 10000
PowerBrick[0].ServoClockDiv = 1

Sys.RtIntPeriod = 0
Sys.ServoPeriod = 1000 * (PowerBrick[0].ServoClockDiv + 1) / PowerBrick[0].PhaseFreq
Sys.PhaseOverServoPeriod = 1 / (PowerBrick[0].ServoClockDiv + 1)

PowerBrick[0].Chan[0].PwmFreqMult = 3
PowerBrick[0].Chan[1].PwmFreqMult = 3
PowerBrick[0].Chan[2].PwmFreqMult = 3
PowerBrick[0].Chan[3].PwmFreqMult = 3
PowerBrick[1].Chan[0].PwmFreqMult = 3
PowerBrick[1].Chan[1].PwmFreqMult = 3
PowerBrick[1].Chan[2].PwmFreqMult = 3
PowerBrick[1].Chan[3].PwmFreqMult = 3

PowerBrick[0].Chan[0].PackOutData = 0
PowerBrick[0].Chan[1].PackOutData = 0
PowerBrick[0].Chan[2].PackOutData = 0
PowerBrick[0].Chan[3].PackOutData = 0
PowerBrick[1].Chan[0].PackOutData = 0
PowerBrick[1].Chan[1].PackOutData = 0
PowerBrick[1].Chan[2].PackOutData = 0
PowerBrick[1].Chan[3].PackOutData = 0

PowerBrick[0].Chan[0].PackInData = 0
PowerBrick[0].Chan[1].PackInData = 0
PowerBrick[0].Chan[2].PackInData = 0
PowerBrick[0].Chan[3].PackInData = 0
PowerBrick[1].Chan[0].PackInData = 0
PowerBrick[1].Chan[1].PackInData = 0
PowerBrick[1].Chan[2].PackInData = 0
PowerBrick[1].Chan[3].PackInData = 0
```

System Gates Sample File for PBL8 Stepper

```
Sys.WpKey = $AAAAAAAA
Sys.pAbortAll = 0
Sys.MaxCoords = 2
Sys.MaxMotors = 9

PowerBrick[1].PhaseFreq = 10000
PowerBrick[1].ServoClockDiv = 1

PowerBrick[0].PhaseFreq = 10000
PowerBrick[0].ServoClockDiv = 1

Sys.RtIntPeriod = 0
Sys.ServoPeriod = 1000 * (PowerBrick[0].ServoClockDiv + 1) / PowerBrick[0].PhaseFreq
Sys.PhaseOverServoPeriod = 1 / (PowerBrick[0].ServoClockDiv + 1)

PowerBrick[0].Chan[0].PwmFreqMult = 3
PowerBrick[0].Chan[1].PwmFreqMult = 3
PowerBrick[0].Chan[2].PwmFreqMult = 3
PowerBrick[0].Chan[3].PwmFreqMult = 3
PowerBrick[1].Chan[0].PwmFreqMult = 3
PowerBrick[1].Chan[1].PwmFreqMult = 3
PowerBrick[1].Chan[2].PwmFreqMult = 3
PowerBrick[1].Chan[3].PwmFreqMult = 3

PowerBrick[0].Chan[0].PackOutData = 0
PowerBrick[0].Chan[1].PackOutData = 0
PowerBrick[0].Chan[2].PackOutData = 0
PowerBrick[0].Chan[3].PackOutData = 0
PowerBrick[1].Chan[0].PackOutData = 0
PowerBrick[1].Chan[1].PackOutData = 0
PowerBrick[1].Chan[2].PackOutData = 0
PowerBrick[1].Chan[3].PackOutData = 0

PowerBrick[0].Chan[0].PackInData = 0
PowerBrick[0].Chan[1].PackInData = 0
PowerBrick[0].Chan[2].PackInData = 0
PowerBrick[0].Chan[3].PackInData = 0
PowerBrick[1].Chan[0].PackInData = 0
PowerBrick[1].Chan[1].PackInData = 0
PowerBrick[1].Chan[2].PackInData = 0
PowerBrick[1].Chan[3].PackInData = 0

BrickLV.Chan[0].TwoPhaseMode = 1
BrickLV.Chan[1].TwoPhaseMode = 1
BrickLV.Chan[2].TwoPhaseMode = 1
BrickLV.Chan[3].TwoPhaseMode = 1
BrickLV.Chan[4].TwoPhaseMode = 1
BrickLV.Chan[5].TwoPhaseMode = 1
BrickLV.Chan[6].TwoPhaseMode = 1
BrickLV.Chan[7].TwoPhaseMode = 1
```

Step 3: Power-On Reset PLC

The Power-on reset PLC serves two purposes:

- Clearing amplifier faults.
- Applying and saving (any) changes made to the **BrickLV** saved structure elements, such as **BrickLV.Chan[].TwoPhaseMode**.

This PLC may already be a part of the Power Brick LV Template in IDE.

Power-On Reset PLC Sample for PBL4

```
OPEN PLC PowerOnResetPLC
Sys.WDTReset = 5000 / (Sys.ServoPeriod * 2.258)
CALL DelayTimer.msec(5)

BrickLV.Reset = 1
CALL DelayTimer.msec(5)

WHILE (BrickLV.Reset == 1){}
IF (BrickLV.Reset == 0)
{
   // HOUSEKEEPING
   PowerBrick[0].Chan[0].CountError = 0
   PowerBrick[0].Chan[1].CountError = 0
   PowerBrick[0].Chan[2].CountError = 0
   PowerBrick[0].Chan[3].CountError = 0

   Sys.MaxPhaseTime = 0
   Sys.MaxServoTime = 0
   Sys.MaxRtIntTime = 0
   Sys.MaxBgTime = 0
   CALL DelayTimer.msec(5)

   // HERE, ENABLE SUBSEQUENT APPLICATION PLCs

   Sys.WDTReset = 0
   DISABLE PLC PowerOnResetPLC
   CALL DelayTimer.msec(5)
}
ELSE
{
   // RESET FAILED? TAKE ACTION
   KILL 1..4
   DISABLE PLC 0,2..31
   SEND 1"BRICK LV RESET FAILED !!!"
   Sys.WDTReset = 0
   DISABLE PLC PowerOnResetPLC
   CALL DelayTimer.msec(5)
}
CLOSE
```

Power-On Reset PLC Sample for PBL8

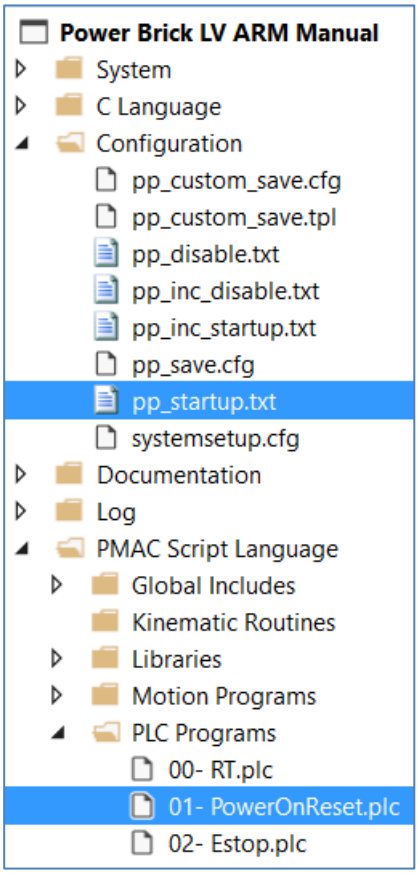
```
OPEN PLC PowerOnResetPLC
Sys.WDTReset = 5000 / (Sys.ServoPeriod * 2.258)
CALL DelayTimer.msec(5)

BrickLV.Reset = 1
CALL DelayTimer.msec(5)

WHILE (BrickLV.Reset == 1){}
IF (BrickLV.Reset == 0)
{
   // HOUSEKEEPING
   PowerBrick[0].Chan[0].CountError = 0
   PowerBrick[0].Chan[1].CountError = 0
   PowerBrick[0].Chan[2].CountError = 0
   PowerBrick[0].Chan[3].CountError = 0
   PowerBrick[1].Chan[0].CountError = 0
   PowerBrick[1].Chan[1].CountError = 0
   PowerBrick[1].Chan[2].CountError = 0
   PowerBrick[1].Chan[3].CountError = 0

   Sys.MaxPhaseTime = 0
   Sys.MaxServoTime = 0
   Sys.MaxRtIntTime = 0
   Sys.MaxBgTime = 0
   CALL DelayTimer.msec(5)

   // HERE, ENABLE SUBSEQUENT APPLICATION PLCs

   Sys.WDTReset = 0
   DISABLE PLC PowerOnResetPLC
   CALL DelayTimer.msec(5)
}
ELSE
{
   // RESET FAILED? TAKE ACTION
   KILL 1..8
   DISABLE PLC 0,2..31
   SEND 1"BRICK LV RESET FAILED !!!"
   Sys.WDTReset = 0
   DISABLE PLC PowerOnResetPLC
   CALL DelayTimer.msec(5)
}
CLOSE
```

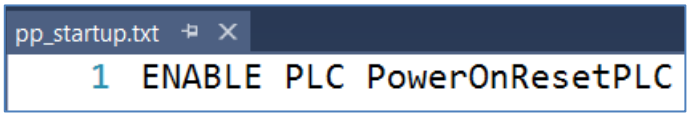
The process of waiting for the **BrickLV.Reset** to execute in a PLC consumes a significant amount of background cycles and risks triggering a foreground soft watchdog fault (**Sys.WDTFault = 1**). Setting **Sys.WDTReset** temporarily to a larger value alleviates this issue.

Note

The **Sys.WDTReset** expression stated in the PLC example should ensure the proper setting regardless of the user specified clock frequencies.

The power-on reset PLC must execute on power-up/reset. This is done by enabling the PLC in the **pp_startup.txt** file under the Configuration folder.





Step 4: Applying Power-On Reset PLC and System Gates Settings

Some System Gates file settings such as clock speeds may require a reset to completely take effect. The Power-On Reset PLC must be run quickly after start up. To solve both of these problems, after changing the global settings in the "System Gates" file and verifying the Power-On Reset PLC will run, do the following.

1. Issue a build and download
2. Save
3. $$$

Step 5: Scaling and Verifying Encoder Feedback

Scaling to Engineering Units

For stepper motors without encoder - using direct micro-stepping technique - this section is skipped.

For all other motor types using an encoder feedback device, this section describes how to verify functionality and scale motor units into engineering units.

Note

This section assumes that the encoder device has been wired and configured properly per the encoder type (e.g. Digital Quadrature, Sinusoidal, etc…) section.

Warning

Motor[].PosSf and **Motor[].Pos2Sf** are part of the motor structure elements, including servo loop. If they are changed, many other elements need to be adjusted such as acceleration, speed settings as well as servo loop gains.

The three motor elements relevant to scaling raw counts into engineering units are:

- **Motor[].PosSf**
- **Motor[].Pos2Sf**
- **Motor[].PosUnit**

Care must be taken, **Motor[].PosSf** and **Motor[].Pos2Sf** are part of the motor structure elements, including servo loop. If they are changed, many other elements need to be adjusted such as acceleration, speed settings as well as servo loop gains.

Motor[].PosUnit changes the unit display in the IDE position window. It does not affect the operation of the motor.

Motor[].PosUnit	Unit Name	Motor[].PosUnit	Unit Name
0	m.u. (motor unit)	8	Mil (in/1000)
1	Count (ct)	9	Revolution
2	Meter (m)	10	Radian (rad)
3	Millimeter (mm)	11	Degree (deg)
4	Micrometer (µm)	12	Gradian (grad)
5	Nanometer (nm)	13	Arcminute (')
6	Picometer (pm)	14	Arcsecond (")
7	Inch (in)		

When **Motor[].PosSf** and **Motor[].Pos2Sf** are changed, all elements described in the Software Reference Manual referring to m.u. (motor unit) now become engineering units instead. For example, if a user changes scaling to mm, **Motor[].JogSpeed** unit is now mm/msec instead of mu/msec. Below, are scaling examples:

Direct drive rotary motor/encoder in degrees

A 23-bit rotary encoder/motor (yielding 8,388,608 counts per revolution) tied directly to the load.

```
GLOBAL Mtr1CtsPerRev = 8388608
GLOBAL Mtr1DegsPerRev = 360
Motor[1].PosSf = Mtr1DegsPerRev / Mtr1CtsPerRev
Motor[1].Pos2Sf = Mtr1DegsPerRev / Mtr1CtsPerRev
Motor[1].PosUnit = 11
```

Geared rotary motor/encoder in inches

A 17-bit rotary encoder/motor (yielding 131,072 counts per revolution) with a 5:1 gear reduction to the load.

```
GLOBAL Mtr1CtsPerRev = 131072
GLOBAL Mtr1RevsPerInch = 5 / 1
Motor[1].PosSf = Mtr1RevsPerInch / Mtr1CtsPerRev
Motor[1].Pos2Sf = Mtr1RevsPerInch / Mtr1CtsPerRev
Motor[1].PosUnit = 7
```

Linear motor/encoder in millimeters

A 1nm BiSS linear encoder scale.

```
GLOBAL Mtr1ResMm = 0.000001
Motor[1].PosSf = Mtr1ResMm
Motor[1].Pos2Sf = Mtr1ResMm
Motor[1].PosUnit = 3
```

Note

Motor[].Pos2Sf is not always equal to **Motor[].PosSf** such as in a dual-feedback system.

Note

For Coordinate System assignments, care must be taken now since the motor is scaled in engineering units and in most cases the scaling would simply be one to one e.g. **&1#1->X**.

Verifying Encoder Feedback


Warning

The absence of encoder data is potentially a very dangerous condition in closed-loop control, because the servo loop no longer has any idea what the true physical position of the motor is – usually it thinks it is "stuck" – and it can react wildly, often causing a runaway condition.

The goal of this section is to verify, before continuing with the motor setup, that the encoder is:

- Counting in both directions of travel
- Reporting the correct distance of rotation/travel

Counting in both directions

This can be done by moving the motor by hand e.g. clockwise, counter-clockwise, positive or negative while monitoring the position window in the IDE.


Note

The user must also check if the position stable (within an inherent dithering amount) at standstill.

For troubleshooting purposes, the user can always look at the “raw count data”. Depending on the type of encoder, the following table shows which register to check:

Encoder Type	Raw Count Register(s)
Digital Quadrature	PowerBrick[].Chan[].ServoCapt
Sinusoidal	PowerBrick[].Chan[].ServoCapt
Sinusoidal ACI	PowerBrick[].Chan[].ServoCapt
Resolver	PowerBrick[].Chan[].ServoCapt
Serial with Gate3	PowerBrick[].Chan[].SerialEncDataA PowerBrick[].Chan[].SerialEncDataB (optional)
Serial with ACC-84B	ACC-84B[].Chan[].SerialEncDataA ACC-84B [].Chan[].SerialEncDataB (optional)

Note, that the primary registers shown above are the source of the corresponding Encoder Conversion Table.

Secondarily, the output of the Encoder Conversion Table itself can be verified using the structure element **EncTable[].PrevEnc**. Note, that **EncTable[].PrevEnc** is not multiplied by **EncTable[].ScaleFactor**.

Reporting the Correct Distance

The user can verify if the feedback device is counting correctly by moving the motor a known amount and recording the elapsed distance shown in the position window in the IDE. In some cases, the **#nHMZ** (where n is the motor number) command can be used to zero the position display.

If the counting is incorrect, make sure that the following is set up correctly:

- **EncTable[].ScaleFactor**
- **Motor[].PosSf**
- **Motor[].Pos2Sf**
- **Motor[].EncType**

Step 6: Motor Setup


Caution

If the +24 VDC abort input is not wired in or disabled in software (**Sys.pAbortAll = 0**), PMAC will try to close the loop on the motor every time it is enabled which could cause the motor the move or jump if it has not been fully set up.

Having performed steps 1 through 4 of the Manual Motor Setup Section, Motor and channel specific parameters can now be configured to finalize the commissioning of a motor by type.


Note

A motor or channel parameter which is not discussed in the structure elements below is assumed – and should typically be left – at default.

All the motor structure elements in subsequent examples of this section refer to a generic Motor[] or Motor[1]. It is the user's responsibility to modify for the appropriate Motor number being set up.

Common Structure Element Settings

Brushless Motor

```
Motor[1].pLimits = PowerBrick[0].Chan[0].Status.a // =0 if limits are not wired
Motor[1].AdcMask = $FFFC0000
Motor[1].AmpFaultLevel = 1
Motor[1].PhaseCtrl = 4
Motor[1].PhaseOffset = 683
```

Stepper Motor with Encoder

```
Motor[1].AdcMask = $FFFC0000
Motor[1].AmpFaultLevel = 1
Motor[1].PhaseOffset = 512
Motor[1].PhaseCtrl = 4
Motor[1].pLimits = PowerBrick[0].Chan[0].Status.a
```

Brushed Motor

```
Motor[1].pLimits = PowerBrick[0].Chan[0].Status.a
Motor[1].AdcMask = $FFFC0000
Motor[1].AmpFaultLevel = 1
Motor[1].PhaseCtrl = 4
Motor[1].PhaseMode = 3
Motor[1].PhaseOffset = 512
Motor[1].PhasePosSf = 0
Motor[1].pAbsPhasePos = Sys.pushm
Motor[1].PowerOnMode = 2
```

Stepper Motor without Encoder – Direct Microstepping

➢ **DIRECT MICRO-STEPPING COUNTS PER REVOLUTION**

With the direct micro-stepping technique, the number of counts per revolution is evaluated using the following equation: Counts/Rev = 360 * 512 / Step Angle

For example, direct micro-stepping of a 1.8° stepper motor produces 102,400 counts (or motor units) per revolution.

➢ **ON-GOING POSITION**

Method 1: Without Scaling

This method is suitable for users who wish to operate the stepper motor in "raw" motor units.

```
GLOBAL Mtr1CtsPerRev = 360 * 512 / 1.8  // User inputs only motor step size e.g. 1.8
Motor[1].PosUnit = 0

EncTable[1].type = 11
EncTable[1].pEnc = Motor[1].PhasePos.a
EncTable[1].index1 = 5
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 255
EncTable[1].index6 = 1
EncTable[1].ScaleFactor = 1 / (256 * (EncTable[1].index5 + 1) * EXP2(EncTable[1].index1))

Motor[1].ServoCtrl = 1
Motor[1].pLimits = PowerBrick[0].Chan[0].Status.a
Motor[1].PhaseCtrl = 6
Motor[1].PhaseMode = 1
Motor[1].PhasePosSf = 0
Motor[1].PhaseOffset = 512
Motor[1].pAbsPhasePos = PowerBrick[0].Chan[0].PhaseCapt.a
Motor[1].PowerOnMode = 2
Motor[1].AdcMask = $FFFC0000
Motor[1].AmpFaultLevel = 1
Motor[1].SlipGain = Sys.PhaseOverServoPeriod / (Motor[1].Stime + 1)
Motor[1].AdvGain = 1/16*Sys.PhaseOverServoPeriod*(0.25/Sys.ServoPeriod/Sys.PhaseOverServoPeriod)

Motor[1].Servo.Kp = 1
Motor[1].Servo.Kvfb = 0
Motor[1].Servo.Kvifb = 0
Motor[1].Servo.Kvff = 1
Motor[1].Servo.Kviff = 0
Motor[1].Servo.Kaff = 1
Motor[1].Servo.Ki = 0
Motor[1].Servo.SwZvInt = 0
```

Method 2: With Scaling to User Engineering Units
This method is suitable for users who wish to operate the stepper motor in user engineering units (e.g. degrees, millimeters).

Example 1: Direct-drive 1.8º stepper motor in degrees

```
GLOBAL Mtr1CtsPerRev = 360 * 512 / 1.8  // User inputs only motor step size e.g. 1.8
GLOBAL Mtr1DegsPerRev = 360
GLOBAL Mtr1CtsPerDeg = Mtr1CtsPerRev / Mtr1DegsPerRev
Motor[1].PosUnit = 11

EncTable[1].type = 11
EncTable[1].pEnc = Motor[1].PhasePos.a
EncTable[1].index1 = 5
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 255
EncTable[1].index6 = 1
EncTable[1].ScaleFactor = 1/(Mtr1CtsPerDeg*256*(EncTable[1].index5+1)*EXP2(EncTable[1].index1))

Motor[1].ServoCtrl = 1
Motor[1].pLimits = PowerBrick[0].Chan[0].Status.a
Motor[1].PhaseCtrl = 6
Motor[1].PhaseMode = 1
Motor[1].PhasePosSf = 0
Motor[1].PhaseOffset = 512
Motor[1].pAbsPhasePos = PowerBrick[0].Chan[0].PhaseCapt.a
Motor[1].PowerOnMode = 2
Motor[1].AdcMask = $FFFC0000
Motor[1].AmpFaultLevel = 1
Motor[1].SlipGain = Sys.PhaseOverServoPeriod / (Motor[1].Stime + 1)
Motor[1].AdvGain = 1/16*Sys.PhaseOverServoPeriod*(0.25/Sys.ServoPeriod/Sys.PhaseOverServoPeriod)

Motor[1].Servo.Kp = Mtr1CtsPerDeg
Motor[1].Servo.Kvfb = 0
Motor[1].Servo.Kvifb = 0
Motor[1].Servo.Kvff = Mtr1CtsPerDeg
Motor[1].Servo.Kviff = 0
Motor[1].Servo.Kaff = Mtr1CtsPerDeg
Motor[1].Servo.Ki = 0
Motor[1].Servo.SwZvInt = 0
```

Example 2: 1:10 gear ratio to load stepper motor in mm

```
GLOBAL Mtr1CtsPerRev = 360 * 512 / 1.8  // User inputs only motor step size e.g. 1.8
GLOBAL Mtr1MmPerRev = 1 / 10            // User input
GLOBAL Mtr1CtsPerMm = Mtr1CtsPerRev / Mtr1MmPerRev
Motor[1].PosUnit = 3

EncTable[1].type = 11
EncTable[1].pEnc = Motor[1].PhasePos.a
EncTable[1].index1 = 5
EncTable[1].index2 = 0
EncTable[1].index3 = 0
EncTable[1].index4 = 0
EncTable[1].index5 = 255
EncTable[1].index6 = 1
EncTable[1].ScaleFactor = 1/(Mtr1CtsPerMm*256*(EncTable[1].index5+1)*EXP2(EncTable[1].index1))

Motor[1].ServoCtrl = 1
Motor[1].pLimits = PowerBrick[0].Chan[0].Status.a
Motor[1].PhaseCtrl = 6
Motor[1].PhaseMode = 1
Motor[1].PhasePosSf = 0
Motor[1].PhaseOffset = 512
Motor[1].pAbsPhasePos = PowerBrick[0].Chan[0].PhaseCapt.a
Motor[1].PowerOnMode = 2
Motor[1].AdcMask = $FFFC0000
Motor[1].AmpFaultLevel = 1
Motor[1].SlipGain = Sys.PhaseOverServoPeriod / (Motor[1].Stime + 1)
Motor[1].AdvGain = 1/16*Sys.PhaseOverServoPeriod*(0.25/Sys.ServoPeriod/Sys.PhaseOverServoPeriod)

Motor[1].Servo.Kp = Mtr1CtsPerMm
Motor[1].Servo.Kvfb = 0
Motor[1].Servo.Kvifb = 0
Motor[1].Servo.Kvff = Mtr1CtsPerMm
Motor[1].Servo.Kviff = 0
Motor[1].Servo.Kaff = Mtr1CtsPerMm
Motor[1].Servo.Ki = 0
Motor[1].Servo.SwZvInt = 0
```

Note

All motor elements (e.g. **Motor[].JogSpeed**) with motor units are now in engineering units instead of counts and must be programmed as such. No other changes are required for the proper operation of the motor.

PWM Scale Factor

For all types of motors, the PWM scale factor specifies the maximum command output (voltage limiter).

- If the motor rated voltage is greater than or equal to ⩾ the input DC voltage:

```
Motor[1].PwmSf = 0.95 * 16384
```

- If the input DC voltage is greater than > the motor rated voltage:

```
GLOBAL DcBusInput = 48          // DC Bus input voltage [VDC] –User Input
GLOBAL Mtr1DCVoltage = 24       // Motor #1 DC rated voltage [VDC] –User Input
Motor[1].PwmSf = 0.95 * 16384 * Mtr1DCVoltage / DcBusInput
```

On-going Phase Position

Stepper Motor without Encoder – Direct Microstepping

Setting up the on-going phase position for stepper motors using the direct micro-stepping technique is not necessary, Motor[].PhasePosSf is already set to 0 in common motor settings section.

Brushed Motor

Setting up the on-going phase position for brushed motors is not necessary, Motor[].PhasePosSf is already set to 0 in common motor settings section.

Stepper Motor with Encoder

The ongoing phase position for stepper motors with encoders is set up similarly to brushless motors. The number of poles pairs (**NoOfPolePairs**) is computed as follows:

NoOfPolePairs = 360 / (Step Angle * 4).

Example: A 1.8° step motors yields 50 pair poles.

Brushless Motor

Following are guidelines for setting up the ongoing phase position **(Motor[].PhasePosSf)** with various types of encoders. Some motor and encoder data sheet information is necessary to compute **Motor[].PhasePosSf** properly:

NoOfPolePairs is the number of pair poles of a rotary motor.
CountsPerRevolution is the number of raw quadrature encoder counts per revolution of a rotary motor.
LinesPerRevolution is the number of sine cycles of a sinusoidal rotary encoder/motor.
ECL$_{mm}$ is the linear motor electrical cycle length or magnetic pitch (e.g. 60.96 mm).
RES$_{mm}$ is the linear encoder resolution (a.k.a. pitch) in the same unit as the ECL (e.g. 1 µm = 0.001 mm).
ResPolePairs is the resolver number of pole pairs.
SingleTurnBits is the number of bits of single turn position data for rotary serial encoder.

- **QUADRATURE ENCODER**

Structure Element	Value
Motor[].pPhaseEnc	PowerBrick[].Chan[].PhaseCapt.a
Motor[].PhaseEncLeftshift	0
Motor[].PhaseEncRightshift	0
Rotary: Motor[].PhasePosSf	2048 * **NoOfPolePairs** / (256 * **CountsPerRevolution**)
Linear: Motor[].PhasePosSf	2048 * **RES$_{mm}$** / (256 * **ECL$_{mm}$**)

- **SINUSOIDAL ENCODER (WITH STANDARD INTERPOLATOR)**

Structure Element	**Value**
Motor[].pPhaseEnc	PowerBrick[].Chan[].PhaseCapt.a
Motor[].PhaseEncLeftshift	0
Motor[].PhaseEncRightshift	0
Rotary: Motor[].PhasePosSf	2048 * **NoOfPolePairs** / (**LinesPerRevolution** * 16384)
Linear: Motor[].PhasePosSf	2048 * $\mathbf{RES_{mm}}$ / (16384 * $\mathbf{ECL_{mm}}$)

- **SINUSOIDAL ENCODER (WITH ACI INTERPOLATOR)**

Structure Element	**Value**
Motor[].pPhaseEnc	PowerBrick[].Chan[].PhaseCapt.a
Motor[].PhaseEncLeftshift	0
Motor[].PhaseEncRightshift	0
Rotary: Motor[].PhasePosSf	2048 * 4 * **NoOfPolePairs** / (**LinesPerRevolution** * 65536)
Linear: Motor[].PhasePosSf	2048 * 4 * $\mathbf{RES_{mm}}$ / (65536 * $\mathbf{ECL_{mm}}$)

- **RESOLVER ENCODER**

Structure Element	**Value**
Motor[].pPhaseEnc	PowerBrick[].Chan[].AtanSumOfSqr.a
Motor[].PhaseEncLeftshift	0
Motor[].PhaseEncRightshift	0
Rotary: Motor[].PhasePosSf	2048 * **NoOfPolePairs** / (**ResPolePairs** * 4294967298)

➢ **SERIAL ENCODER WITH GATE3**

For ongoing phase position, it is simplest to process only the portion of single-turn position data that is available in **PowerBrick[].Chan[].SerialEncDataA**. This will not limit the resolution or hinder the performance.

Motor[].PhaseEncRightshift is set to the number of unwanted bits to the right of the desired data, so that a right shift can be performed to clear that unwanted data. **Motor[].PhaseEncLeftshift** is then set to the number of bits the data must be shifted left (after the right shift) to make the Most Significant Bit (MSB) of your position data bit #31.

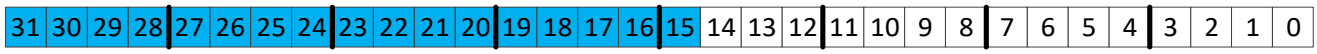
Structure Element	**Value**
Motor[].pPhaseEnc	PowerBrick[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift	Number of bits to left shift (second operation)
Motor[].PhaseEncRightshift	Number of bits to right shift (first operation)
Rotary: Motor[].PhasePosSf	2048 * **NoOfPolePairs** / $2^{(\text{PhaseEncLeftshift} + \text{SingleTurnBits})}$
Linear: Motor[].PhasePosSf	2048 * $\textbf{RES}_{\textbf{mm}}$ / ($\textbf{ECL}_{\textbf{mm}}$ * $2^{\text{PhaseEncLeftshift}}$)

Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of **SerialEncDataA**.

PowerBrick[].Chan[].SerialEncDataA

Shift left 15 bits to MSB for rollover.

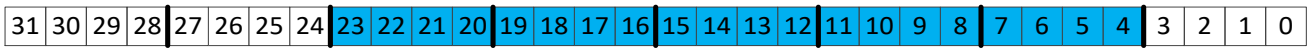
After Shifting



Motor[].pPhaseEnc = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 15
Motor[].PhaseEncRightshift = 0
Rotary: Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(15 + 17)}$
Linear: Motor[].PhasePosSf = 2048 * $\textbf{RES}_{\textbf{mm}}$ / ($\textbf{ECL}_{\textbf{mm}}$ * 2^{15})

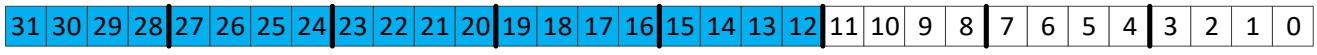
Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.

PowerBrick[].Chan[].SerialEncDataA



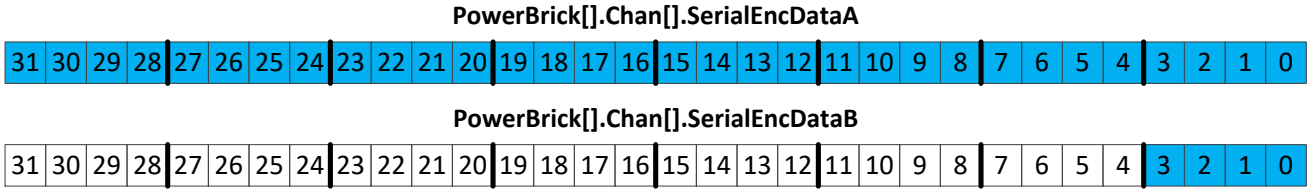
Shift right 4 bits first to get rid of unwanted data. Shift left 12 bits to MSB for rollover.

After Shifting

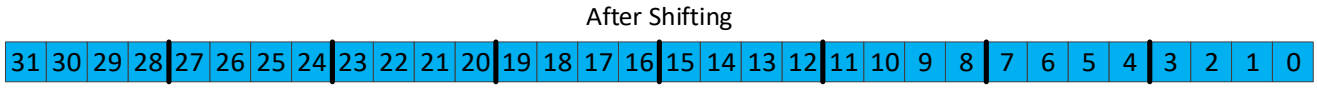


Motor[].pPhaseEnc = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 12
Motor[].PhaseEncRightshift = 4
Rotary: Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(12 + 20)}$
Linear: Motor[].PhasePosSf = 2048 * $\textbf{RES}_{\textbf{mm}}$ / ($\textbf{ECL}_{\textbf{mm}}$ * 2^{12})

Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of **SerialEncDataA** and extending to bit #3 of **SerialEncDataB**.

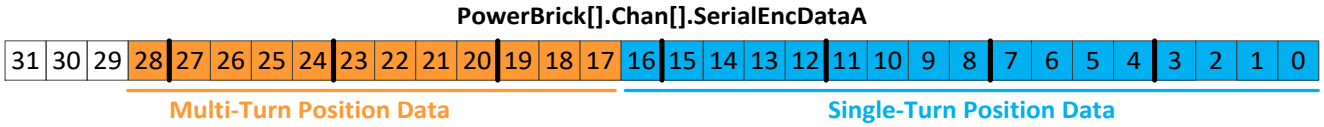


No shifting is required.

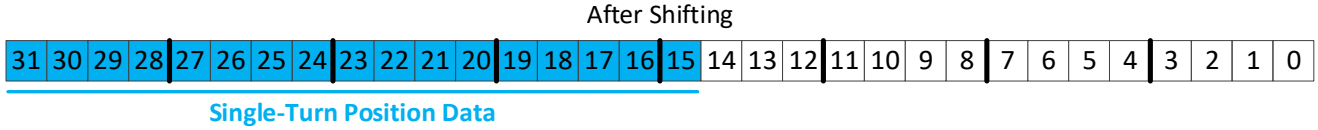


Motor[].pPhaseEnc = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 0
Motor[].PhaseEncRightshift = 0
Rotary: Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(0 + 32)}$
Linear: Motor[].PhasePosSf = 2048 * $\mathbf{RES_{mm}}$ / ($\mathbf{ECL_{mm}}$ * 2^0)

Example 4: A 29-bit binary serial encoder with 17 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA**.

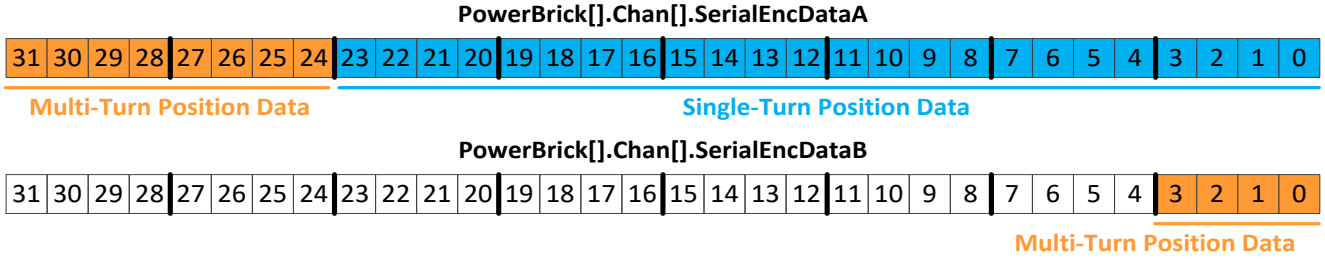


Shift left 15 bits to MSB for rollover.

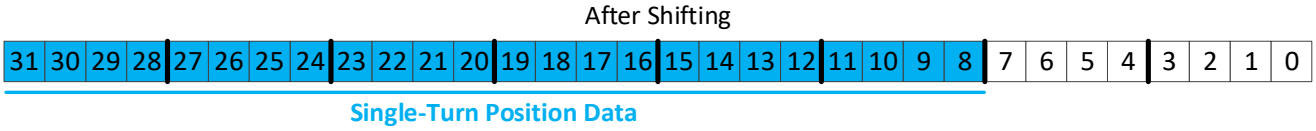


Motor[].pPhaseEnc = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 15
Motor[].PhaseEncRightshift = 0
Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(15 + 17)}$

Example 5: A 36-bit binary serial encoder with 24 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA** and continuously extending to bit #3 of **SerialEncDataB**.



Shift left 8 bits to MSB for rollover.



Motor[].pPhaseEnc = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 8
Motor[].PhaseEncRightshift = 0
Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(8 + 24)}$

Note

The **Motor[].PhasePosSf** is best entered as an expression (e.g. a ratio of integers) to let the Power PMAC calculate the exact value.

➢ SERIAL ENCODER WITH ACC-84B

For ongoing phase position, it is simplest to process only the portion of single-turn position data that is available in **PowerBrick[].Chan[].SerialEncDataA**. This will not limit the resolution or hinder the performance.

Motor[].PhaseEncRightshift is set to the number of unwanted bits to the right of the desired data, so that a right shift can be performed to clear that unwanted data. **Motor[].PhaseEncLeftshift** is then set to the number of bits the data must be shifted left (after the right shift) to make the Most Significant Bit (MSB) of your position data bit #31.

Although data may appear to start at bit 0 in the script environment, internally it is only 24 bits starting at bit 8. This means data should be right shifted 8 bits more than would be expected from viewing **Acc84B[].Chan[].SerialEncDataA** in the watch window or terminal.

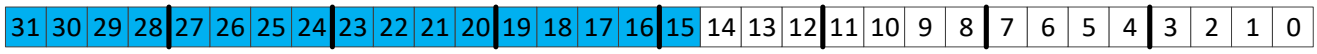
Structure Element	Value
Motor[].pPhaseEnc	ACC84B[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift	Number of bits to left shift (second operation)
Motor[].PhaseEncRightshift	Number of bits to right shift (first operation)
Rotary: Motor[].PhasePosSf	2048 * **NoOfPolePairs** / $2^{(\textbf{PhaseEncLeftshift} + \textbf{SingleTurnBits})}$
Linear: Motor[].PhasePosSf	2048 * $\textbf{RES}_{\textbf{mm}}$ / ($\textbf{ECL}_{\textbf{mm}}$ * $2^{\textbf{PhaseEncLeftshift}}$)

Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA**.

ACC84B[].Chan[].SerialEncDataA

Shift right 8 bits first to get rid of unwanted data. Shift left 15 bits to MSB for rollover.

After Shifting



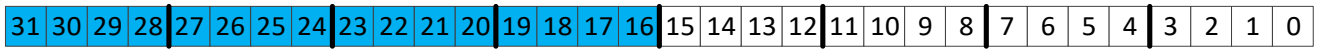
Motor[].pPhaseEnc = ACC84B[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 15
Motor[].PhaseEncRightshift = 8
Rotary: Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(15 + 17)}$
Linear: Motor[].PhasePosSf = 2048 * $\textbf{RES}_{\textbf{mm}}$ / ($\textbf{ECL}_{\textbf{mm}}$ * 2^{15})

Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of the 24 bit **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.

ACC84B[].Chan[].SerialEncDataA

Shift right 12 bits first to get rid of unwanted data. Shift left 16 bits to MSB for rollover.

After Shifting



Motor[].pPhaseEnc = ACC84B[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 16
Motor[].PhaseEncRightshift = 12
Rotary: Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(16 + 16)}$
Linear: Motor[].PhasePosSf = 2048 * $\mathbf{RES_{mm}}$ / ($\mathbf{ECL_{mm}}$ * 2^{16})

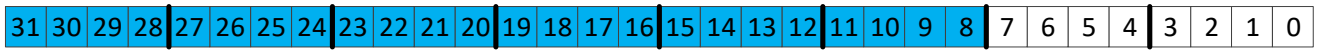
Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA** and extending to bit #11 of **SerialEncDataB**.

ACC84B[].Chan[].SerialEncDataA

ACC84B[].Chan[].SerialEncDataB

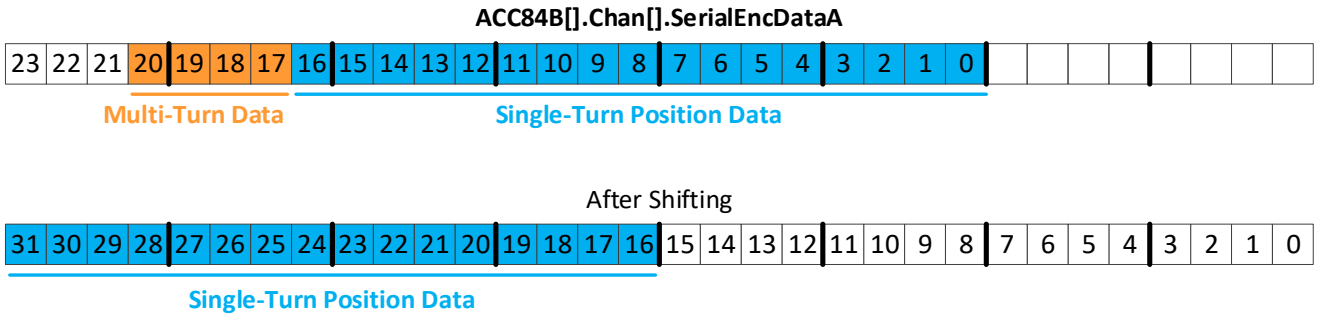
Shift right 8 bits first to get rid of unwanted data. Shift left 8 bits to MSB for rollover.

After Shifting



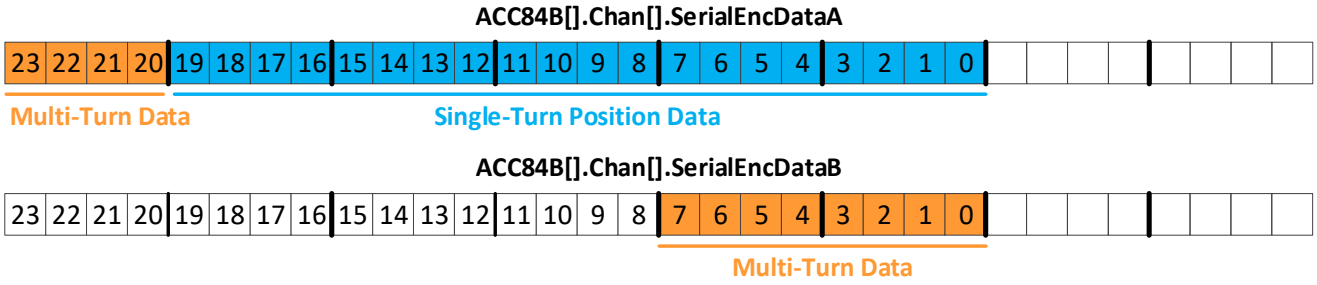
Motor[].pPhaseEnc = ACC84B[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 8
Motor[].PhaseEncRightshift = 8
Rotary: Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(8 + 36)}$
Linear: Motor[].PhasePosSf = 2048 * $\mathbf{RES_{mm}}$ / ($\mathbf{ECL_{mm}}$ *2^{8})

Example 4: A 21-bit binary serial encoder with 17 bits of single-turn and 4 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA**.

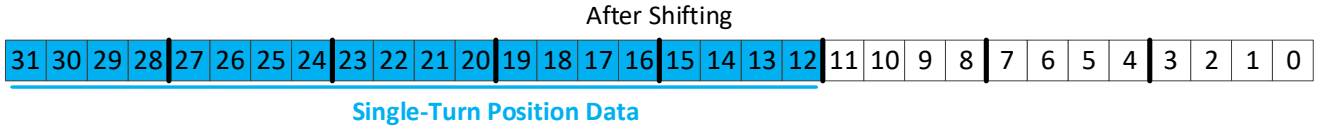


Motor[].pPhaseEnc = ACC84B[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 15
Motor[].PhaseEncRightshift = 8
Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(15 + 17)}$

Example 5: A 32-bit binary serial encoder with 20 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA** and continuously extending to bit #7 of **SerialEncDataB**.



Shift right 8 bits first to get rid of unwanted data. Shift left 12 bits to Single Turn MSB for rollover.



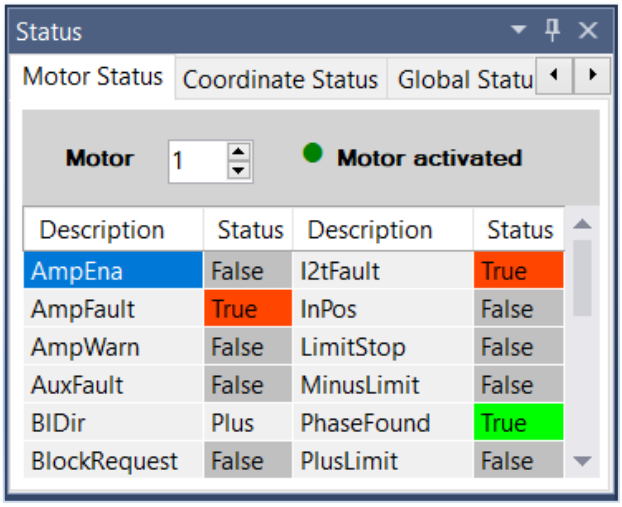
Motor[].pPhaseEnc = ACC84B[].Chan[].SerialEncDataA.a
Motor[].PhaseEncLeftshift = 12
Motor[].PhaseEncRightshift = 8
Motor[].PhasePosSf = 2048 * **NoOfPolePairs** / $2^{(12 + 20)}$

Note

The **Motor[].PhasePosSf** is best entered as an expression (e.g. a ratio of integers) to let the Power PMAC calculate the exact value.

I2T Protection

The Power Brick LV can be set up to fault a motor if the time-integrated current levels exceed a certain threshold. This can protect the motor (and drive) from damage due to overheating. It integrates the square of current over time – commonly known as I2T ("eye-squared-tee") protection.



For maximum protection, the Power PMAC performs the I2T calculations even when the motor is killed. In normal operation, measured currents should be very near zero in the killed state, and this is not important. However, it is possible during initial setup that incorrect settings cause Power PMAC to detect high current values, and it may take some time even after the settings have been corrected for the integrated values to "decay" to permit the amplifier to be enabled.

When an I2T fault occurs, the motor is killed, the amplifier fault and I2tFault bits are set (as seen in the motor status window in the IDE software). These bits can be accessed using the motor structure elements **Motor[].AmpFault** and **Motor[].I2TFault**.

The stricter current specifications (lower) between the motor and the Power Brick LV channel should be used in the I2T calculations:

Peak Current Limit			**Continuous Current Limit**	
Current rating	**Value to use**	**Time at peak**	**Current rating**	**Value to use**
Motor < Drive	That of Motor	That of motor	Motor < Drive	That of Motor
Motor > Drive	That of Drive	That of drive (1 second)	Motor > Drive	That of Drive

The max ADC, or full current reading, is specified by the power rating of the channel:

Channel Rating	**Max ADC**
0.25 A / 0.75 A	1.6925 A
1 A / 3 A	6.770 A
5 A / 15 A	33.85 A

Note

Power PMAC's I2T is a motor thermal protection feature; the Power Brick LV amplifier(s) has its own built-in I2T model which protects the power transistors.

Stepper Motor without Encoder – Direct Micro-stepping

➢ **MAXIMUM CURRENT**

Example 1: Motor current limits given in RMS values or using those of the Power Brick LV

```
GLOBAL Ch1MaxAdc = 33.85                  // Max ADC reading [A peak]      --User Input
GLOBAL Ch1RmsPeakCur = 7.2                // RMS Peak Current [A rms]      --User Input
GLOBAL Ch1RmsContCur = 3.6                // RMS Continuous Current [A rms]--User Input
GLOBAL Ch1TimeAtPeak = 1                  // Time Allowed at peak [sec]    --User Input
```

```
GLOBAL Ch1TempMaxDac = Ch1RmsPeakCur * 32768 * SQRT(2) / Ch1MaxAdc
Motor[1].I2TSet = Ch1RmsContCur * 32768 * SQRT(2) / Ch1MaxAdc
Motor[1].I2tTrip = (POW(Ch1TempMaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak
```

Example 2: Motor current limits given in peak values

```
GLOBAL Ch1MaxAdc = 33.85                  // Max ADC reading [A peak]      --User Input
GLOBAL Ch1PeakCur = 7.2                   // Peak Current [A peak]         --User Input
GLOBAL Ch1ContCur = 3.6                   // Continuous Current [A peak]   --User Input
GLOBAL Ch1TimeAtPeak = 1                  // Time Allowed at peak [sec]    --User Input
```

```
GLOBAL Ch1TempMaxDac = Ch1PeakCur * 32768 / Ch1MaxAdc
Motor[1].I2TSet = Ch1ContCur * 32768 / Ch1MaxAdc
Motor[1].I2tTrip = (POW(Ch1TempMaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak
```

Caution

If the current limits of the motor are given as peak values, there is no need to multiply by √2 (1.414).

➢ **MAXIMUM COMMAND OUTPUT / SPEED LIMIT**

The maximum command output, **Motor[].MaxDac**, represents the speed limit. After calculating **Motor[].MaxDac** (in I2T section), compute its product with the slip gain. In order to achieve the nominal speed, this product must be less than or equal to 512.

(Motor[].MaxDac * Motor[].SlipGain) must be **≤ 512** to achieve max speed

If the product is greater than 512, and the application requires reaching top speeds then the servo frequency must be reduced (or **Motor[].Stime** increased) until this condition is satisfied. **Example:**

```
GLOBAL Mtr1MaxRpm = 1500    // Motor Maximum Speed [RPM]      --User Input
GLOBAL Mtr1StepAngle = 1.8  // Motor Step Angle [degrees]     --User Input
Motor[1].MaxDac =  Mtr1MaxRpm / 60000 * (360 / (4 * Mtr1StepAngle)) * 2048 *  Sys.ServoPeriod
```

➢ **MAXIMUM ACHIEVABLE SPEEDS**

The direct micro-stepping technique has a maximum speed of 1024 micro-steps per servo cycle, and 512 micro-steps per phase cycle.

Example: For a standard 100-pole (1.8°) stepper motor with a 5 kHz servo update rate, and a 20 kHz phase update rate. The maximum achievable speed can be computed as follows. A 1.8° full-step motor has a 4 x 1.8° = 7.2° commutation cycle

Servo limitation:

$$\text{MaxRpm} = 1{,}024\ \frac{\mu\text{step}}{\text{cycle}} \times 5{,}000\ \frac{\text{cycle}}{\text{sec}} \times \frac{7.2}{2{,}048}\ \frac{\text{deg}}{\mu\text{step}} \times \frac{1}{360}\ \frac{\text{rev}}{\text{deg}} \times 60\ \frac{\text{sec}}{\text{min}} = 3{,}000\ \text{rpm}$$

Phase Limitation:

$$\text{MaxRpm} = 512\ \frac{\mu\text{step}}{\text{cycle}} \times 20{,}000\ \frac{\text{cycle}}{\text{sec}} \times \frac{7.2}{2{,}048}\ \frac{\text{deg}}{\mu\text{step}} \times \frac{1}{360}\ \frac{\text{rev}}{\text{deg}} \times 60\ \frac{\text{sec}}{\text{min}} = 6{,}000\ \text{rpm}$$

Therefore, the maximum achievable speed (servo limitation) is 3,000 rpm. Higher speeds will require increasing the update rate(s) correspondingly.

Note

Few users will operate stepper motors at these speeds, but these limits should be calculated and update rates set high enough that desired speeds can be reached.

Stepper Motor with Encoder

Example 1: Motor current limits given in RMS values or using those of the Power Brick LV

```
GLOBAL Ch1MaxAdc = 33.85                  // Max ADC reading [A peak]      --User Input
GLOBAL Ch1RmsPeakCur = 7.2                // RMS Peak Current [A rms]      --User Input
GLOBAL Ch1RmsContCur = 3.6                // RMS Continuous Current [A rms]--User Input
GLOBAL Ch1TimeAtPeak = 1                  // Time Allowed at peak [sec]    --User Input
```

```
Motor[1].MaxDac = Ch1RmsPeakCur * 32768 * SQRT(2) / Ch1MaxAdc
Motor[1].I2TSet = Ch1RmsContCur * 32768 * SQRT(2) / Ch1MaxAdc
Motor[1].I2tTrip = (POW(Motor[1].MaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak
```

Example 2: Motor current limits given in peak values

```
GLOBAL Ch1MaxAdc = 33.85                  // Max ADC reading [A peak]      --User Input
GLOBAL Ch1PeakCur = 7.2                   // Peak Current [A peak]         --User Input
GLOBAL Ch1ContCur = 3.6                   // Continuous Current [A peak]   --User Input
GLOBAL Ch1TimeAtPeak = 1                  // Time Allowed at peak [sec]    --User Input
```

```
Motor[1].MaxDac = Ch1PeakCur * 32768 / Ch1MaxAdc
Motor[1].I2TSet = Ch1ContCur * 32768 / Ch1MaxAdc
Motor[1].I2tTrip = (POW(Motor[1].MaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak
```

Caution

If the current limits of the motor are given as peak values, there is no need to multiply by √2 (1.414).

Brushless Motor

Example 1: Motor current limits given in RMS values or using those of the Power Brick LV

```
GLOBAL Ch1MaxAdc = 33.85                    // Max ADC reading [A peak]      --User Input
GLOBAL Ch1RmsPeakCur = 7.2                  // RMS Peak Current [A rms]      --User Input
GLOBAL Ch1RmsContCur = 3.6                  // RMS Continuous Current [A rms]--User Input
GLOBAL Ch1TimeAtPeak = 1                    // Time Allowed at peak [sec]    --User Input
```

```
Motor[1].MaxDac = Ch1RmsPeakCur * 32768 * SQRT(2) * COSD(30) / Ch1MaxAdc
Motor[1].I2TSet = Ch1RmsContCur * 32768 * SQRT(2) * COSD(30) / Ch1MaxAdc
Motor[1].I2tTrip = (POW(Motor[1].MaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak
```

Example 2: Motor current limits given in peak values

```
GLOBAL Ch1MaxAdc = 33.85                    // Max ADC reading [A peak]      --User Input
GLOBAL Ch1PeakCur = 7.2                     // Peak Current [A peak]         --User Input
GLOBAL Ch1ContCur = 3.6                     // Continuous Current [A peak]   --User Input
GLOBAL Ch1TimeAtPeak = 1                    // Time Allowed at peak [sec]    --User Input
```

```
Motor[1].MaxDac = Ch1PeakCur * 32768 * COSD(30) / Ch1MaxAdc
Motor[1].I2TSet = Ch1ContCur * 32768 * COSD(30) / Ch1MaxAdc
Motor[1].I2tTrip = (POW(Motor[1].MaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak
```

Caution

If the current limits of the motor are given as peak values, there is no need to multiply by √2 (1.414).

Brushed Motor

```
GLOBAL Ch1MaxAdc = 6.770                    // Max ADC reading [A peak]      --User Input
GLOBAL Ch1PeakCur = 2.92                    // Peak Current [A peak]         --User Input
GLOBAL Ch1ContCur = 0.75                    // Continuous Current [A peak]   --User Input
GLOBAL Ch1TimeAtPeak = 1                    // Time Allowed at peak [sec]    --User Input
```

```
Motor[1].MaxDac = Ch1PeakCur * 32768 / Ch1MaxAdc
Motor[1].I2TSet = Ch1ContCur * 32768 / Ch1MaxAdc
Motor[1].I2tTrip = (POW(Motor[1].MaxDac,2) - POW(Motor[1].I2TSet,2)) * Ch1TimeAtPeak
```

Direct Magnetization Current

Brushed Motor

Magnetization current is not necessary for brushed motors.

Stepper Motor with Encoder

Magnetization current is not necessary for stepper motors with encoder.

Brushless Motor

Magnetization current is not necessary for brushless motors.

Stepper Motor Without Encoder – Direct Micro-stepping

The magnetization current is the desired direct current to be introduced into the stepper motor. The higher the mag current, the stiffer the motor holds position. This is at the expense of thermal heating. Typically, the range of **Motor[].IdCmd** is [**Motor[].I2TSet/2** – **Motor[].I2TSet**]. **Example:**

```
Motor[1].IdCmd = Motor[1].I2TSet / 2
```

Note

To get the full torque out of your stepper motor, use the rated current as continuous for I2T Protection and set I2TSet to the high end of this range.

Current Loop Tuning

Stepper Motor Without Encoder – Direct Micro-stepping

Current loop tuning for stepper motors using the direct micro-stepping technique is performed similarly to brushless motors.

Stepper Motor with Encoder

Current loop tuning for stepper motors with encoder is performed similarly to brushless motors.

Brushed Motor

Current loop tuning for brushed motors is performed similarly to brushless motors. However, the IDE tuning software injects "direct" current to perform a current loop step response. To use this tool successfully with DC brush motors:

- **Motor[].PhaseTableBias** must be set manually to ±512 (90° electrical angle) so that direct current corresponds to A-phase current. The sign of ±512 is typically chosen so that a positive step response is produced.
- **Motor[].PhaseMode** must be set to 1 so that bit 1 is zero forcing the Id integrator to be on during tuning.

Note

Remember to set **Motor[].PhaseTableBias** back to 0, and **Motor[].PhaseMode** to 3 for normal motor operation.

Brushless Motor

Current loop tuning is typically performed using the tuning tool in the IDE software.

Note

With some basic knowledge of motor and amplifier parameters, it is possible to calculate the current-loop gains empirically. This is described in the Power PMAC User manual.

The "Simple Auto-tune" and "Auto-tune" tools are straight forward tools which may be used effectively.

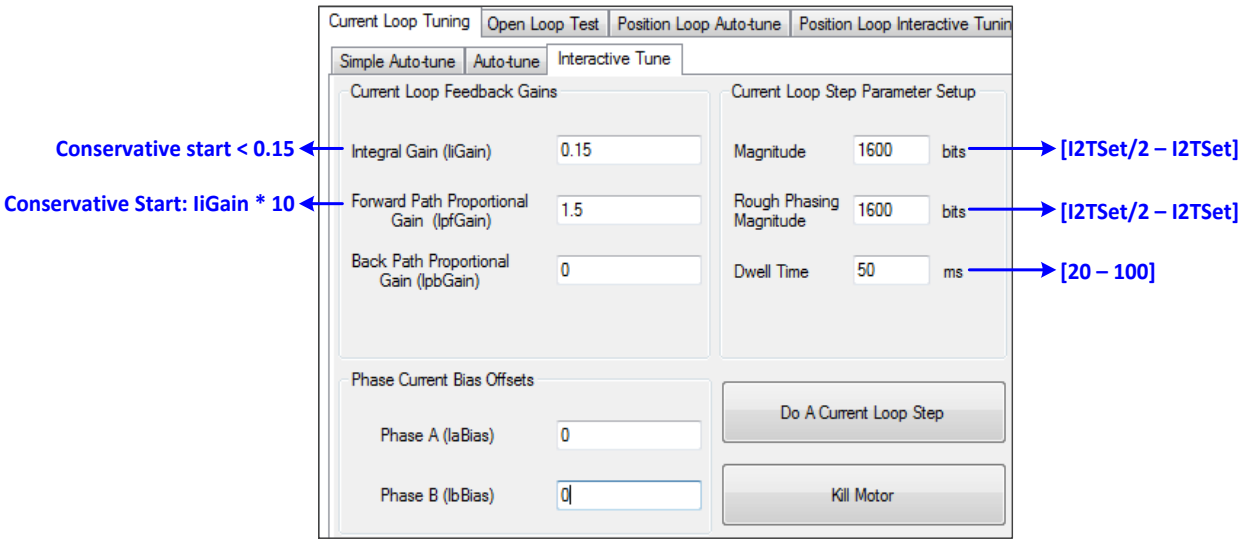
Following, is a practical description of the "Interactive tune" utility.

The current loop step test magnitude and rough phasing are typically in the range of:

Motor[].I2TSet / 2 < Magnitude < **Motor[].I2TSet**

This allows enough current to overcome static non-linear components for a good response without the risk of overheating the motor or triggering an over-current fault.

The "Dwell Time" is typically in the 50 – 100 msec range. This may be extended for slower response motors (high inductance).



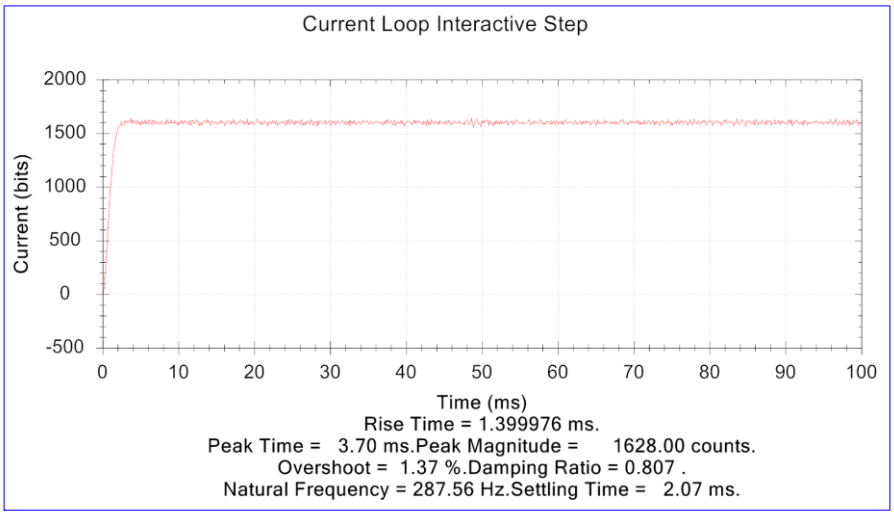
Brushless motors' current loop can be, virtually, tuned using exclusively **Motor[].IiGain** and **Motor[].IpfGain**. In the Power PMAC digital current loop algorithm these gains can be thought of as:

Motor[].IiGain: The transient effort (in reality integral gain).
Motor[].IpfGain: The damping gain (in reality forward path proportional gain).

Motor[].IpbGain can be optionally used in conjunction with **Motor[].IpfGain.**

Current-Loop response with natural Frequencies in the range of 200 – 500 Hz and a rise time of about 1 msec are adequate for most applications. With higher performance motors (e.g. linear), the current loop's natural frequency can be pushed higher. However, tightening the current loop with a lower performance system could have deteriorating effects on the overall position closed-loop performance.

An acceptable current-loop step response should look like:



Establishing Phase Reference

Brushed Motor

Establishing phase reference is not necessary for brushed motors.

Stepper Motor without Encoder – Direct Microstepping

Establishing phase reference is not necessary for stepper motors using the direct micro-stepping technique.

Stepper Motor with Encoder

Establishing phase reference for stepper motors with encoder is performed similarly to brushless motors.

Brushless Motor

When commutating a synchronous multi-phase motor such as a permanent-magnet brushless servo motor, the commutation algorithm must know the absolute position of the rotor within a single commutation cycle so it knows the magnetic field orientation of the rotor. The process of establishing this absolute position sense is known as "phase referencing" or "phasing".

Warning

An unreliable phasing search method can lead to a runaway condition. Test the phasing search method carefully to make sure it works properly under all conceivable conditions, and various locations of the travel. Make sure the fatal following error limit **Motor[].FatalFeLimit** is active and as tight as possible so the motor will be killed quickly in the event of a serious phasing search error.

Setting up a new motor/encoder requires performing an automatic or manual phase referencing routine. In the absence of an absolute phase reference reporting device such as digital hall sensors or absolute encoder, this phase referencing routine may be saved and implemented permanently.

With digital hall sensors, absolute power-on phasing can be configured to perform the phase referencing without the need of moving or energizing the motor. A manual force phasing routine is used to correct for hall sensors' phasing error one time per motor/encoder setup.

With absolute encoders, absolute power-on phasing can be configured to perform the phase referencing without the need of moving or energizing the motor. A manual force phasing routine is used to compute the absolute phase offset one time per motor/encoder setup.

Note

The available torque from a motor is directly proportional to the accuracy of the phase reference. The better the phasing is the less torque loss, current dissipation, and motor/drive thermal losses are.

Note

For best performance, the initial phasing routine (any method) should be done on an unloaded/uncoupled motor.

Note

Vertical axes phasing may require higher output current to overcome gravity, it is strongly advised to implement a balancing mechanism (e.g. weight, air) for these cases.

The following phasing methods are discussed in this section:

- Automatic Stepper Phasing
- Manual "Force" Phasing
- Custom "PLC" Phasing

Choosing a phasing method depends on the feedback device used with the brushless motor. The following table is a summary of the suggested phasing method to use with respect to each type of feedback device:

Type of Feedback Device	**Initial Phasing / Getting Started**	**Final Implementation / Saved Configuration**
Quadrature / Sinusoidal – No Halls	Stepper / Manual	Stepper / Manual / PLC
Quadrature / Sinusoidal – With Halls		Absolute Phasing. Halls phasing correction recommended.
Resolver		Absolute Phasing.
Serial Incremental		Stepper / Manual / PLC
Serial Absolute		Absolute Phasing.

➢ AUTOMATIC STEPPER PHASING

The automatic Stepper phasing technique is one of two phase referencing routines built into the Power PMAC firmware (the other one is the four-guess technique, not discussed here). The automatic stepper method can be used with any type of feedback device. It is simple to set up and can establish a very accurate phase reference.

Without the presence of digital hall sensors or an absolute encoder, the automatic stepper method can be saved and used in the power-up routine of the motor. Prior to implementing it permanently, it is highly recommended to test the automatic stepper method for consistency at random locations of the travel. Setting up the automatic stepper phasing technique requires configuring the following motor structure elements:

- **Motor[].PhaseFindingDac** specifies the magnitude of the output (current) used in the search move. **Motor[].I2Tset / 2** is a "good" conservative value to start with.
- **Motor[].PhaseFindingTime** specifies the amount of time (in real time interrupts) allowed for the search move. This can be computed in milliseconds, per the example equation below.
- **Motor[].AbsPhasePosOffset** specifies the minimum motion that qualifies the search as being a valid search. Typically set to 1/5th of a commutation cycle (2048 / 5).
- **Motor[].PowerOnMode** specifies whether a search move is applied on power-up. This is not advised with the automatic stepper phasing since the main bus power may not be available when the PMAC powers up. Leave bit 1 = 0.

Caution

The Stepper phasing technique is a search operation which requires the motor to move, typically in small steps. Nevertheless, caution should be taken.

Example

```
GLOBAL Mtr1PhasingTime = 3000                    // Total phasing time [msec] --User Input
Motor[1].PhaseFindingTime = Mtr1PhasingTime * 0.5 / (Sys.ServoPeriod * (Sys.RtIntPeriod + 1))
Motor[1].PhaseFindingDac = Motor[1].I2TSet / 2   // Phasing search magnitude  --User Input
Motor[1].AbsPhasePosOffset = 2048 / 5            // Qualifying motor movement
```

Note

The computed **Motor[].PhaseFindingTime** must be greater than 255 and less than 32,768 for the proper implementation of the automatic stepper phasing technique.

Issuing a **#n$** or setting **Motor[].PhaseFindingStep = 1** launches the stepper phasing search move. The pass/fail of the operation is reported by the motor status **Motor[].PhaseFound** bit. If the phasing fails (**Motor[].PhaseFound = 0**) repeatedly:

- Try increasing the magnitude, **Motor[].PhaseFindingDac**.
- Try extending the time allowed for phasing, **Motor[].PhaseFindingTime**.
- Try reversing the encoder decode **PowerBrick[].Chan[].EncCtrl** (e.g. 7 to 3 or vice versa). Not applicable to serial encoders.
- Try swapping two of the motor leads.
- Decouple the motor from the load and try again.

➢ MANUAL "FORCE" PHASING

The manual phasing method consists of locking up the motor tightly onto the zero position of the commutation cycle by forcing current into the offset of its B phase. This manual phasing works with any type of feedback device. It is particularly useful in:

- Establishing a phase reference manually.
- Troubleshooting phasing difficulties.
- Finding the absolute phase offset with absolute serial encoders.

Caution

The manual phasing technique is a search operation which requires the motor to move, typically in small steps. Nevertheless, caution should be taken.

Note

The tighter the motor is locked, the better is the phase reference.

Following, are the basic steps for performing a manual "force" phasing:

1. Make sure the motor is killed and steady.
2. Set **Motor[].IbBias** to a value corresponding to the amount of current to force into the phase. A conservative start is = **Motor[].I2TSet / 2.**
3. Issue a **#nOut0** (where n is the motor number). The motor should lock into a position and exhibit some stiffness when trying to move it by hand.

   Increase **Motor[].IbBias** as necessary until the motor is locked tightly. Exceeding the value of **Motor[].I2TSet** indicates that there is a problem with the amplifier output or that the motor or drive is not sized properly for the load.

   Wait for the motor to settle. In some instances, it may oscillate for an extended amount of time. Some motors may be small enough to safely stabilize by hand.

4. Zero the phase position register if performing a phasing routine; **Motor[].PhasePos = 0**. Or record the corresponding serial data for finding the absolute phase offset with absolute serial encoders.
5. Kill the motor; **#nK**.
6. Reset **Motor[].IbBias = 0**
7. Set the phase found status bit; **Motor[].PhaseFound = 1** if performing a phasing routine.

The motor should be phased at this point, and could be verified with an open loop test. Below are a few troubleshooting tips in case of difficulty:

- Try increasing the magnitude of **Motor[].IbBias.**
- Try reversing the encoder decode **PowerBrick[].Chan[].EncCtrl** (e.g. 7 to 3 or vice versa). Not applicable to serial encoders.
- Try swapping two of the motor leads.
- Decouple the motor from the load, and try again.

➢ **CUSTOM "PLC" PHASING**

Some system may require a more specialized phasing technique due to uneven loads or friction along the travel. This manual phasing PLC may be more desirable for advanced users due to flexibility and more customization capabilities.

This travel distance should theoretically correspond to 1/6 of a commutation cycle size (in motor/encoder units). This is checked against at the end of the routine, and recorded in a pass/fail flag.

- **MtrxPhasingMag** is the amount of current to use for step phasing the motor. Conservative starting estimate **Motor[].I2TSet / 2**.
- **MtrxPhaseAPos** is the actual position of the motor when locked on to phase A.
- **MtrxPhaseBPos** is the actual position of the motor when locked on to phase B.
- **MtrxPhasingDis** is the displacement during the phasing routine.
- **MtrxDisThres** is the minimum travel indicating a successful phasing. 5^{th} of a commutation cycle = 2048 * **EncTable[].ScaleFactor** / (5 * **Motor[].PhasePosSf**)
- **MtrxPhasingPass** is a flag indicating the pass or fail of the phasing routine. =1 pass, =0 fail.

Note

If the motor does not settle between lock-ups, increase the delay time. The threshold with which the filtered velocity is compared to may need to be tweaked as well.

Note

It is highly advised to test the motor phasing with the stepper or manual force phasing method before attempting to use a custom PLC.

```
GLOBAL Mtr1PhasingMag = Motor[1].I2TSet
GLOBAL Mtr1PhaseAPos = 0
GLOBAL Mtr1PhaseBPos = 0
GLOBAL Mtr1PhasingDis = 0
GLOBAL Mtr1DisThres = 2048 * EncTable[1].ScaleFactor / (5 * Motor[1].PhasePosSf)
GLOBAL Mtr1PhasingPass = 0

OPEN PLC CustomPhasingPLC
Mtr1PhasingPass = 0
Motor[1].PhaseFound = 0
Motor[1].IaBias = 0 Motor[1].IbBias = 0
COUT 1:0
CALL DelayTimer.msec(100)
WHILE (ABS(Motor[1].FltrVel) > 5){}

WHILE (Motor[1].IaBias !> Mtr1PhasingMag)
{
    Motor[1].IaBias += 1   Motor[1].IbBias = 0
    CALL DelayTimer.msec(1)
}
CALL DelayTimer.sec(2)
WHILE (ABS(Motor[1].FltrVel) > 5){}
Mtr1PhaseAPos = ABS(Motor[1].ActPos - Motor[1].HomePos)
CALL DelayTimer.msec(250)

WHILE (Motor[1].IbBias !> Mtr1PhasingMag)
{
    Motor[1].IaBias -= 1   Motor[1].IbBias += 1
    CALL DelayTimer.msec(1)
}
CALL DelayTimer.sec(2)
WHILE (ABS(Motor[1].FltrVel) > 5){}
Mtr1PhaseBPos = ABS(Motor[1].ActPos - Motor[1].HomePos)
CALL DelayTimer.msec(250)

Mtr1PhasingDis = ABS(Mtr1PhaseBPos - Mtr1PhaseAPos)
IF(Mtr1PhasingDis >= Mtr1DisThres)
{
    Motor[1].PhasePos = 0
    Motor[1].PhaseFound = 1
    Mtr1PhasingPass = 1
}
ELSE
{
   Mtr1PhasingPass = 0
}
CALL DelayTimer.msec(250)

KILL 1
Motor[1].IaBias  = 0 Motor[1].IbBias  = 0
DISABLE PLC CustomPhasingPLC
CLOSE
```

Open Loop Test

Stepper Motor without Encoder – Direct Microstepping

The open loop test is not necessary for stepper motors using the direct micro-stepping technique.

Stepper Motor with Encoder

The open loop test for stepper motors with encoder is performed similarly to brushless motors.

Brushed Motor

The open loop test for brushed motors with encoder is performed similarly to brushless motors.

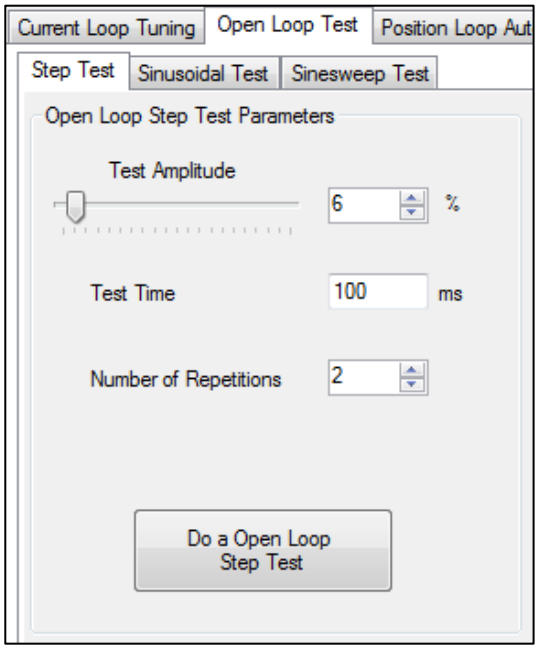
Brushless Motor

The open loop test is a critical step in verifying the proper implementation of the:

- Current loop
- Commutation
- Encoder decode/sense
- Encoder functionality

The open loop test can be executed using the open-loop test tab in the tuning utility in the IDE software.

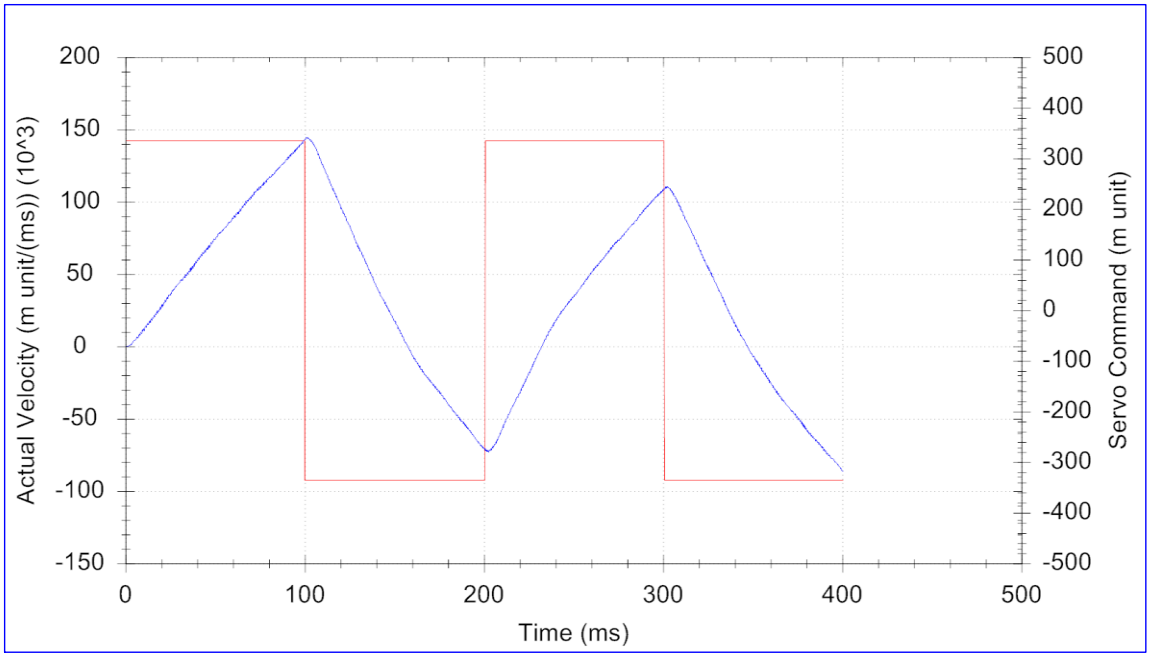
The test amplitude depends on the load/gearing of the motor. Conservative values between 1 – 10% are good starting estimates. The test time is typically under 500 msec, nominally 100 msec. The number of repetitions is user configurable and may depend on the allowed amount of travel.



Caution

Do not attempt to close the position loop on a motor which open loop test has not passed, or shows an inverted saw tooth velocity. This may lead to dangerous runaway conditions.

A positive command should create a velocity and position counting in the positive direction; a negative command should create a velocity and position counting in the negative direction. This is typically observed in the response plot as a velocity saw tooth. A successful open-loop test response looks like:



➢ **TROUBLESHOOTING TIPS:**

The open loop test can fail in two ways:

- Motor cogs to a phase (locks up)
- Plot shows an inverted saw tooth.

This indicates that one or a combination of the following:

- Incorrect commutation cycle size; review **Motor[].PhasePosSf**.
- Reversed encoder direction sense; review **PowerBrick[].Chan[].EncCtrl** (e.g. 7 to 3).
  Not applicable to serial encoders.
- Phasing was not preformed successfully; phase and try again.
- Reversed commutation direction; can be reversed in two ways:
  - Swapping any two of the motor leads
  - Setting **Motor[].PwmSf**, and **Motor[]PhaseOffset** simultaneously to the opposite sign

Position Loop Tuning

Stepper Motor without Encoder – Direct Microstepping

Position loop tuning is not necessary for stepper motors using the direct micro-stepping technique.

Stepper Motor with Encoder

Position loop tuning of stepper motors with encoder is performed similarly to brushless motors.

Brushed Motor

Position loop tuning of brushed motors is performed similarly to brushless motors.

Brushless Motor

Position loop tuning is performed using the tuning utility in the IDE Software.

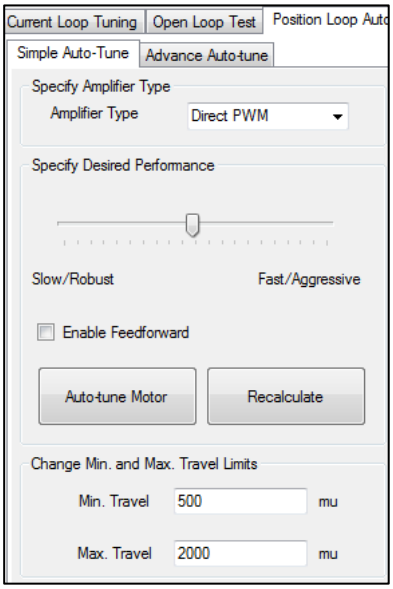
Caution

Do not attempt to close the position loop or perform position loop tuning on a motor which open-loop test has failed. This may lead to dangerous runaway conditions.

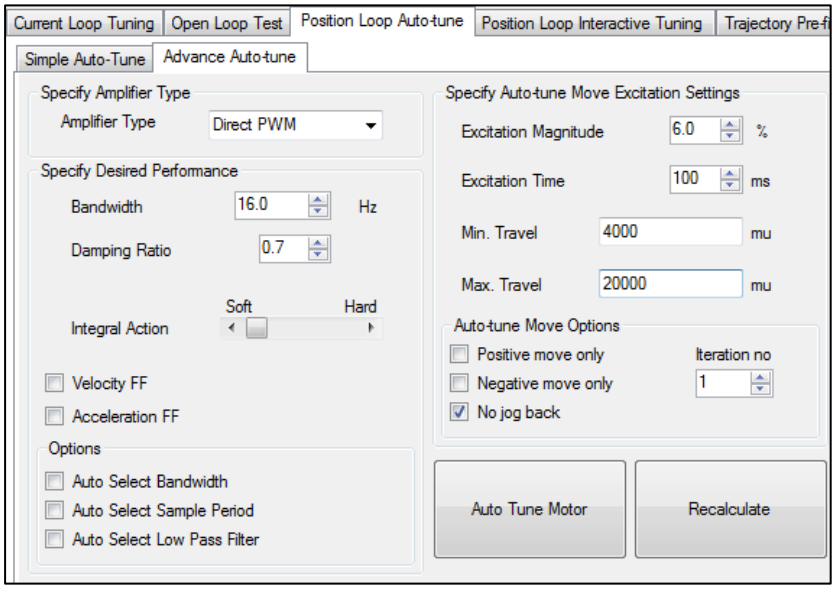
There are three main tuning sub-utilities in the tuning tool:

- Simple auto-tune.
- Advanced Auto-Tune.
- Position-Loop interactive tuning.

Simple Auto-tune



Advanced Auto-tune



For brushless motors, with the Power Brick LV, the amplifier type is always set to PWM.

The **simple auto-tune** is self-explanatory; move the slide left for a slower natural frequency and right for a higher natural frequency. Checking the "enable feedforward" box will also estimate the feedforward gains. This tuning technique may be more suitable for lightly loaded motors, and lower resolution encoders.

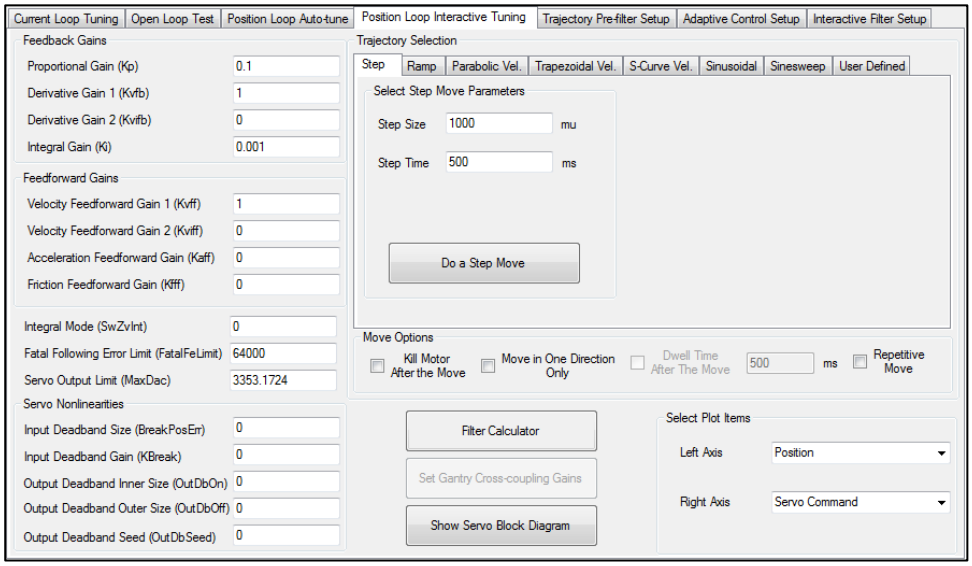
The **advanced auto-tune** introduces more user specific inputs, such as specifying the desired natural frequency, damping ratio, and integral action. The excitation magnitude and time are typically the same as the ones used successfully in the open-loop test.

Note

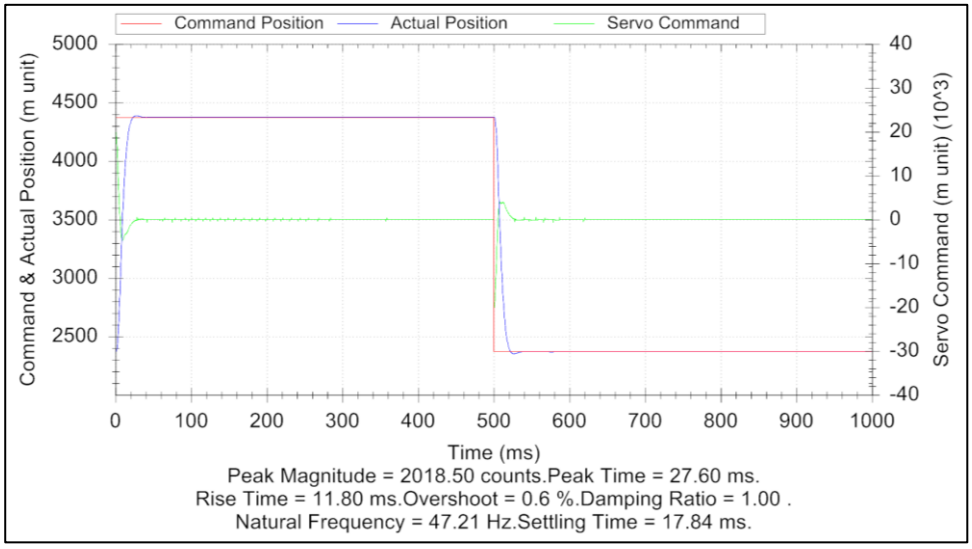
The automatic tuning techniques are conceived for rough tuning, which may be suitable for most applications. Fine tuning is typically performed using the interactive utility.

The **Position-loop interactive tuning** is the fully fletched tuning interface, introducing all the gains used in the servo algorithm, various pre-configured command profiles, and filter tools. The two most common move profiles used in tuning are Step and Parabolic.

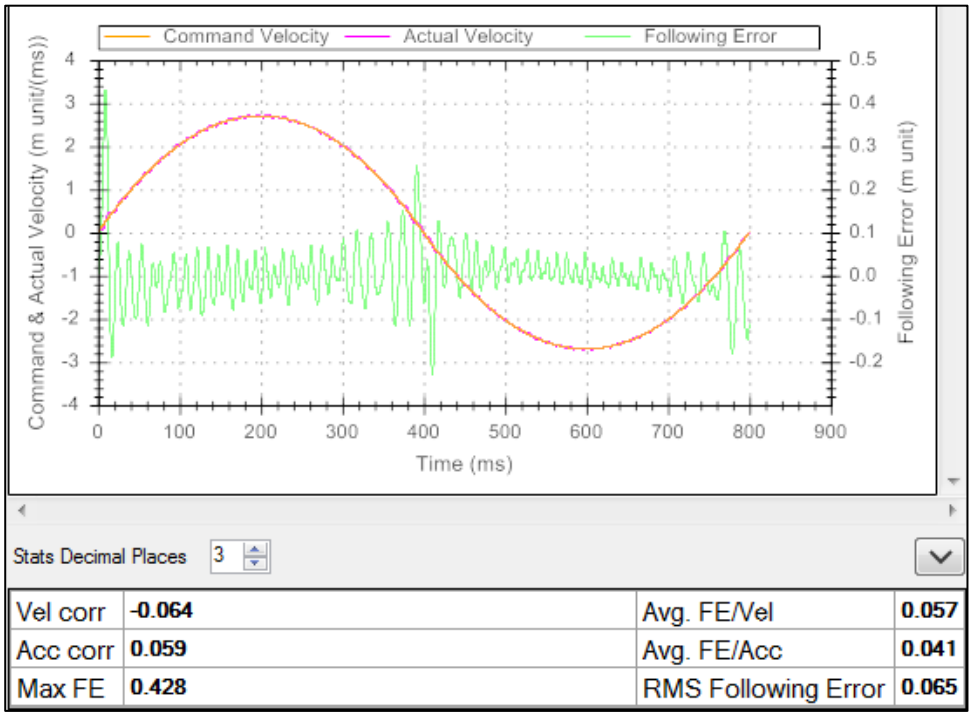
Interactive Tuning



An acceptable step move response would look like:



And an acceptable parabolic move response would look like:



Vel corr	**-0.064**	Avg. FE/Vel	**0.057**
Acc corr	**0.059**	Avg. FE/Acc	**0.041**
Max FE	**0.428**	RMS Following Error	**0.065**

Desirable characteristics to note: following error (green curve above) centered about 0 with minimal amplitude.

Note

With higher resolution encoders, the **Motor[].Servo.MaxPosErr** may need to be set to a higher than the default value allowing larger position error in the servo filter.

Absolute Power-on Phasing

Stepper Motor without Encoder – Direct Microstepping

The absolute power-on phasing is not necessary as per this section for stepper motors using the direct micro-stepping technique.

Brushed Motor

The absolute power-on phasing is not necessary as per this section for brushed motors.

Stepper Motor with Encoder

The absolute power-on phasing for stepper motors with encoder is performed similarly to brushless motors.

Brushless Motor

Absolute power-on phasing is configurable with feedback devices providing an absolute reference capability; devices such as hall sensors, resolvers, or absolute serial encoders.

The absolute power-on phasing allows the phasing (figuring out the commutation rotor-angle position) of a motor without the need of a search move (motion) or energizing the motor.

With the 4 key motor structure elements (described in the examples below) configured and saved, issuing a **#n$** or **Motor[].PhaseFindingStep = 1** will initiate the absolute phasing computation.

A successful operation sets the **Motor[].PhaseFound** bit of the motor status to 1.

Alternately, automatic power-on absolute phasing can be configured (and saved) by setting bit #1 of the motor structure element **PowerOnMode.**

Example: Motor[].PowerOnMode = Motor[].PowerOnMode | $2.

Note

If the encoder power (5 VDC) is supplied from the X1 – X8 connectors, then the encoder is ensured to receive power by the time the PMAC boots up. However, if the encoder power is wired external, the user must ensure that this supply is turned on by the time the PMAC boots up and before phasing.

Note

Another bit of **Motor[].PowerOnMode** may be used to perform an automatic **HMZ** command.

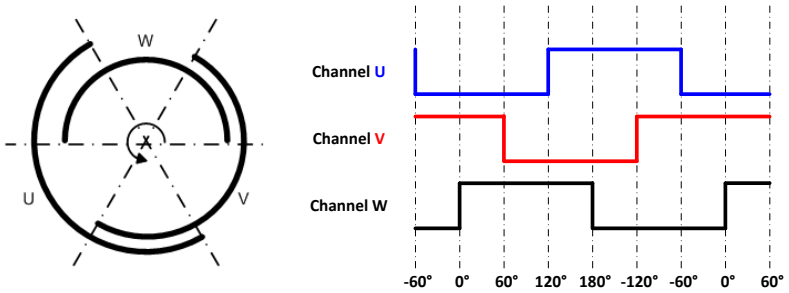
➢ HALL EFFECT PHASING

Digital Hall Effect sensors can be used for computing a rough absolute phase reference on power-up without the need for a phasing search move. They provide absolute information about where the motor is positioned with respect to its commutation cycle. They are desirable because, just like with absolute encoders, the motor can be phased on power-up without any movement.

Note

Inherently, digital hall sensors have an error of about ±30°, resulting in a loss of torque of about 15%. This should be corrected (fine phasing) for top operation.

The Power Brick LV supports both the conventional 120°, and less common 60° spacing. This section focuses on the more standard 120° spacing, each signal nominally with 50% duty cycle, and nominally 1/3 cycle apart.



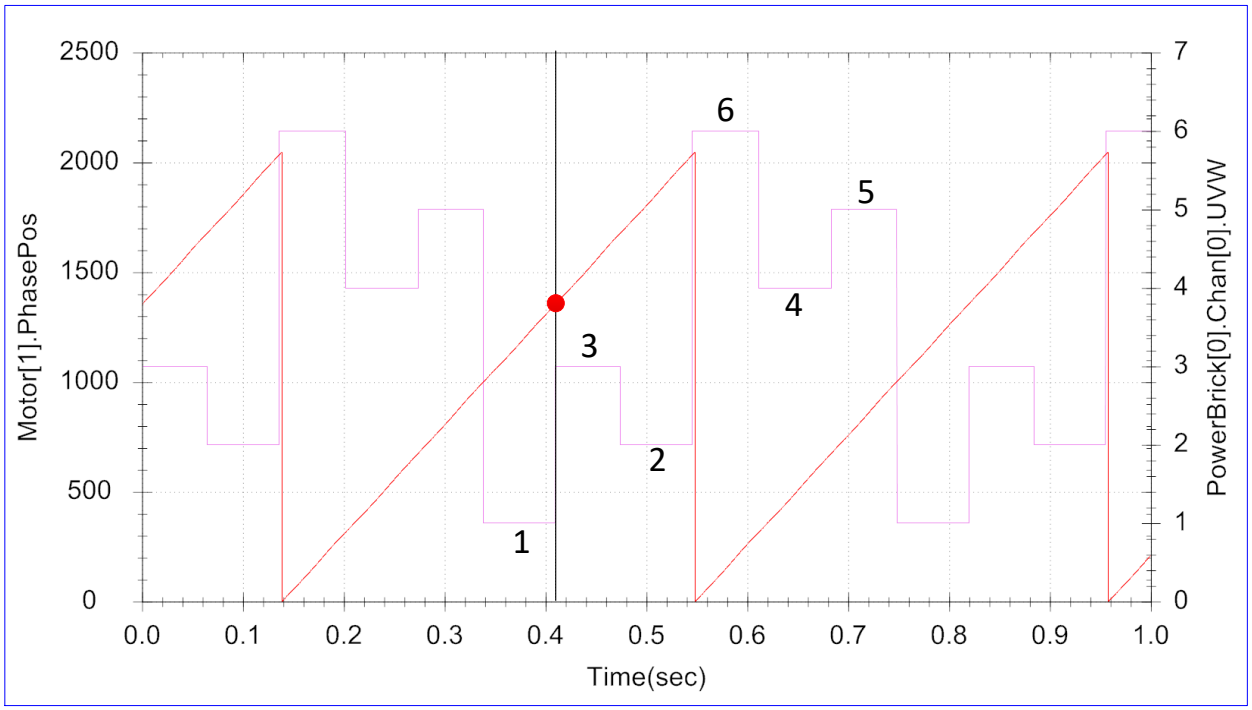
Setting up digital Hall Effect sensors' absolute phasing requires:

- The motor to be phased initially (using the stepper/manual technique)
- Moving the motor either by hand or with jog commands.
  Moving the motor by hand with geared or loaded motors may not be possible. In these cases, it is recommended to perform the open loop test and rough position loop tuning first then come back for setting up the Hall sensors.

The key motor structure elements necessary for configuring Hall sensors' absolute phasing are:

- **Motor[].pAbsPhasePos** = PowerBrick[].Chan[].Status.a
- **Motor[].AbsPhasePosFormat** = $400030C (always for halls 120° spacing)
- **Motor[].AbsPhasePosSf.** The **Motor[].AbsPhasePosSf** reflects the direction sense of the halls with respect to the commutation counting direction. This is the UVW transition when moving the motor in the positive direction of the encoder:
  = 2048 / 12 if the **PowerBrick[].Chan[].UVW** transition is from 1 to 3
  = -2048 / 12 if the **PowerBrick[].Chan[].UVW** transition is from 3 to 1
- **Motor[].AbsPhasePosOffset**
  The **Motor[].AbsPhasePosOffset** is the phase position at that transition.

These settings can be configured using the plot utility in the IDE software. Moving or jogging the motor by hand in the **positive direction** while gathering **Motor[].PhasePos** and the corresponding **PowerBrick[].Chan[].UVW** should produce the following:



Motor[].AbsPhasePosSF =	2048 / 12	If the transition is 1-3
	–2048 / 12	If the transition is 3-1

Motor[].AbsPhasePosOffset is equal to **Motor[].PhasePos** at the transition.

Example:

```
Motor[1].pAbsPhasePos = PowerBrick[0].Chan[0].Status.a
Motor[1].AbsPhasePosFormat = $400030C
Motor[1].AbsPhasePosSF = 2048 / 12  // --UserInput
Motor[1].AbsPhasePosOffset = 1362   // --UserInput
```

➢ GENERATING HALL EFFECT PHASING PARAMETERS USING A PLC

Alternately, the following PLC example configures the **Motor[].AbsPhasePosSf**, and **Motor[].AbsPhasePosOffset** automatically.

- Enable the PLC
- Move the motor at a slow to average speed (by hand or using jog commands) in the **positive direction** of the encoder.
- Once **Motor[].AbsPhasePosOffset** is posted, your Halls settings are finished. Discard the PLC and save the four key motor structure parameters in the project as well as in the PMAC.

Example:

```
PTR Ch1Halls->PowerBrick[0].Chan[0].UVW

OPEN PLC HallsPLC
Motor[1].AbsPhasePosSF = 0
Motor[1].AbsPhasePosOffset = 0

// Check Direction
WHILE (Motor[1].AbsPhasePosSF == 0)
{
    IF (Ch1Halls == 1)
    {
        WHILE (Ch1Halls == 1){}
        IF (Ch1Halls == 3) {Motor[1].AbsPhasePosSF = 2048 / 12}
        ELSE {Motor[1].AbsPhasePosSF = -2048 / 12}
    }
}

// Capture Motor[].PhasePos at Transition
WHILE (Motor[1].AbsPhasePosOffset == 0)
{
    IF (Motor[1].AbsPhasePosSF > 0 && Ch1Halls == 1 && Motor[1].AbsPhasePosOffset == 0)
    {
        WHILE (Ch1Halls == 1){}
        Motor[1].AbsPhasePosOffset = Motor[1].PhasePos
    }
    IF (Motor[1].AbsPhasePosSF < 0 && Ch1Halls == 3 && Motor[1].AbsPhasePosOffset == 0)
    {
        WHILE (Ch1Halls == 3){}
        Motor[1].AbsPhasePosOffset = Motor[1].PhasePos
    }
}
DISABLE PLC HallsPLC
CLOSE
```

➢ **HALL PHASING CORRECTION (FINE PHASING)**

Inherently, digital hall sensors have an error of about ±30° resulting in a loss of torque of about 15%. Correcting for hall sensors' error can be achieved with a simple procedure. For better efficiency, this correction is strongly recommended for all applications using hall sensors for "absolute" phasing.

The hall phasing correction requires homing the motor. If the motor's position loop has not been tuned for closed loop commands it may be more practical, after phasing with the stepper/manual technique, to carry on to the open loop test and position loop tuning then come back for hall phasing correction.

Note

Hall phasing correction requires homing the motor.

The following are the necessary steps to implement the hall phasing correction:

1. Phase the motor, as best as possible, using the stepper / manual technique.
2. Home the motor to a reliable reference; encoder index or combination of flag and index.
   Not to be changed after the initial installation.
3. Record **Motor[].PhasePos**.
   This value can be saved in **Motor[].AbsPhasePosForce**.

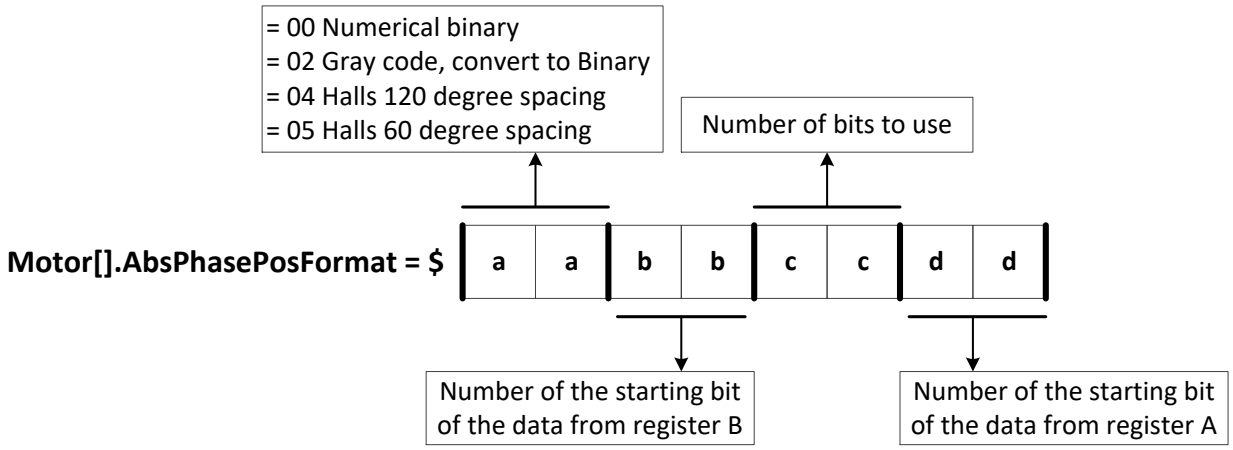
After saving **Motor[].AbsPhasePosForce** in the project and the PMAC, and on the next power cycle:

- Phase the motor using halls by issuing **#n$** or **Motor[].PhaseFindingStep = 1**
- Home the motor to the same reference used in the phase correction routine.
- Once homed and settled, set **Motor[].PhasePos = Motor[].AbsPhasePosForce**.
  The hall phasing correction is now complete.

➢ ABSOLUTE SERIAL ENCODER PHASING WITH GATE3

With absolute serial encoders, the four key elements for setting up absolute phasing are:

- **Motor[].pAbsPhasePos = PowerBrick[].Chan[].SerialEncDataA.a**
- **Motor[].AbsPhasePosFormat**
  - Encoders with no multi-turn position data are unsigned. Rotary encoders with multi-turn position data are signed.
  - Only 32 bits of position data can be used for absolute phasing. This should be the upper 32 bits of (single-turn) positon data.



- **Motor[].AbsPhasePosSf**
  If less than 32 **SingleTurnBits**
  Rotary: = 2048 * **NoOfPolesPairs** / $2^{\mathbf{SingleTurnBits}}$
  Linear: = 2048 * $\mathbf{RES_{mm}}$ / $\mathbf{ECL_{mm}}$
  If more than 32 **SingleTurnBits**
  Rotary: = 2048 * **NoOfPolesPairs** / 2^{32}
  Linear: = 2048 * $\mathbf{RES_{mm}}$ * $2^{\mathbf{SingleTurnBits-32}}$ / $\mathbf{ECL_{mm}}$

- **Motor[].AbsPhasePosOffset = –PhaseForceTest * Motor[].AbsPhasePosSf**
  Where:
  - **NoOfPolePairs** is the number of pole pairs of a rotary motor
  - **SingleTurnBits** is the number of bits of single turn position data for rotary serial encoder.
  - $\mathbf{ECL_{mm}}$ is the linear motor electrical cycle length or magnetic pitch (e.g. 60.96 mm)
  - $\mathbf{RES_{mm}}$ is the linear encoder resolution in the same unit as the ECL (e.g. 1 µm = 0.001 mm)
  - **PhaseforceTest** is the position value recorded from the Stepper Phasing Force Test.

Note

Only 32 bits of position data can be used for absolute phasing.

Note

Gray code conversion should be omitted here if it had been already implemented in **PowerBrick[].Chan[].SerialEncCmd**.

Stepper Phasing Force Test

The following, are the basic steps for performing the stepper phasing force test, which is similar to manual motor phasing:

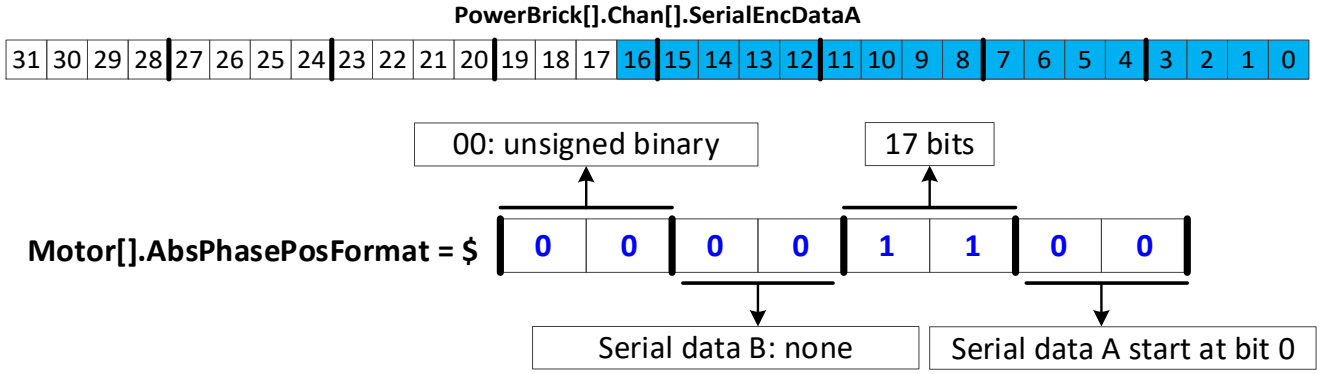
1. Make sure the motor is killed and steady.
2. Set **Motor[].IbBias** to a value corresponding to the amount of current to force into the phase. A conservative start is = **Motor[].I2TSet / 2.**
3. Issue a **#nOut0** (where n is the motor number). The motor should lock into a position and exhibit some stiffness when trying to move it by hand.

   Increase **Motor[].IbBias** as necessary until the motor is locked tightly. Exceeding the value of **Motor[].I2TSet** indicates that there is a problem with the amplifier output or that the motor or drive is not sized properly for the load.

   Wait for the motor to settle. In some instances, it may oscillate for an extended amount of time. Some motors may be small enough to safely stabilize by hand.

4. Record the entire position from Serial Data registers. See examples below for **PhaseForceTest** equations with masking and shifting.
5. Kill the motor; **#nK**.
6. Reset **Motor[].IbBias = 0**

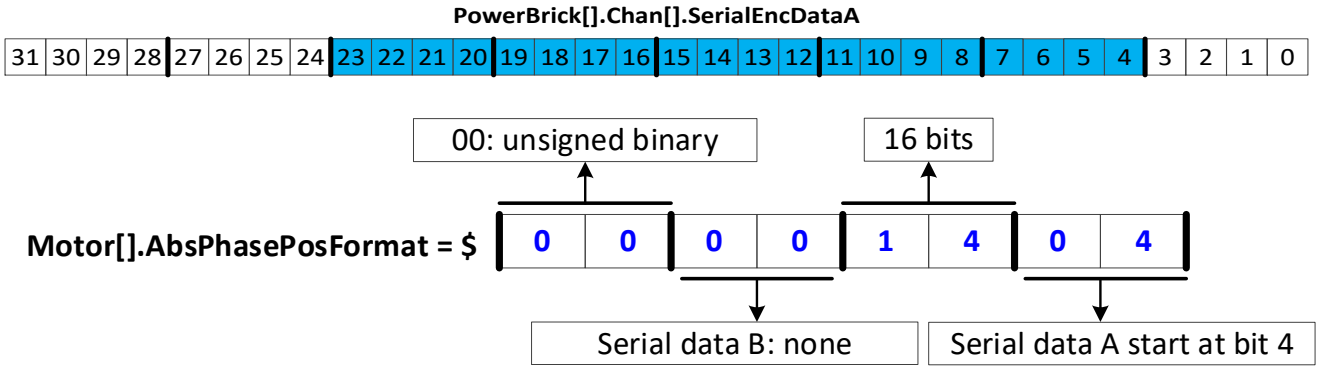
Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of **SerialEncDataA**.



Motor[].pAbsPhasePos = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00001100
Rotary: Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{17}
Linear: Motor[].AbsPhasePosSf = 2048 * $\mathbf{RES_{mm}}$ / $\mathbf{ECL_{mm}}$
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = PowerBrick[].Chan[].SerialEncDataA & $0001FFFF L0

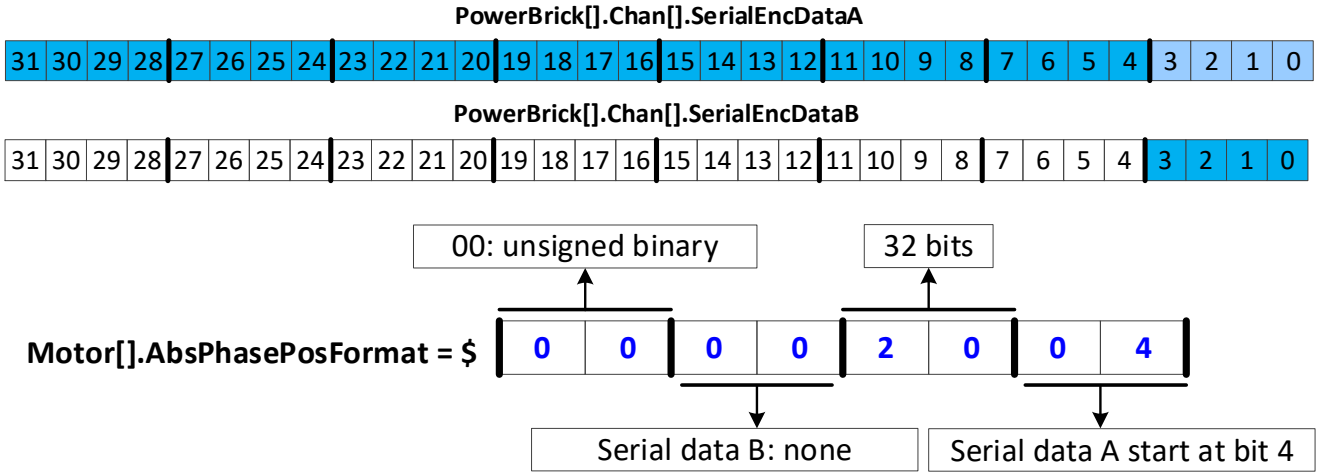
Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.



Motor[].pAbsPhasePos = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00001404
Rotary: Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{20}
Linear: Motor[].AbsPhasePosSf = 2048 * $\mathbf{RES_{mm}}$ / $\mathbf{ECL_{mm}}$
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = (PowerBrick[].Chan[].SerialEncDataA & $00FFFFF0) >> L0

Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of **SerialEncDataA** and extending to bit #3 of **SerialEncDataB**. We will use the upper 32 bits; that is the maximum allowed number of bits for the power-on absolute commutation.



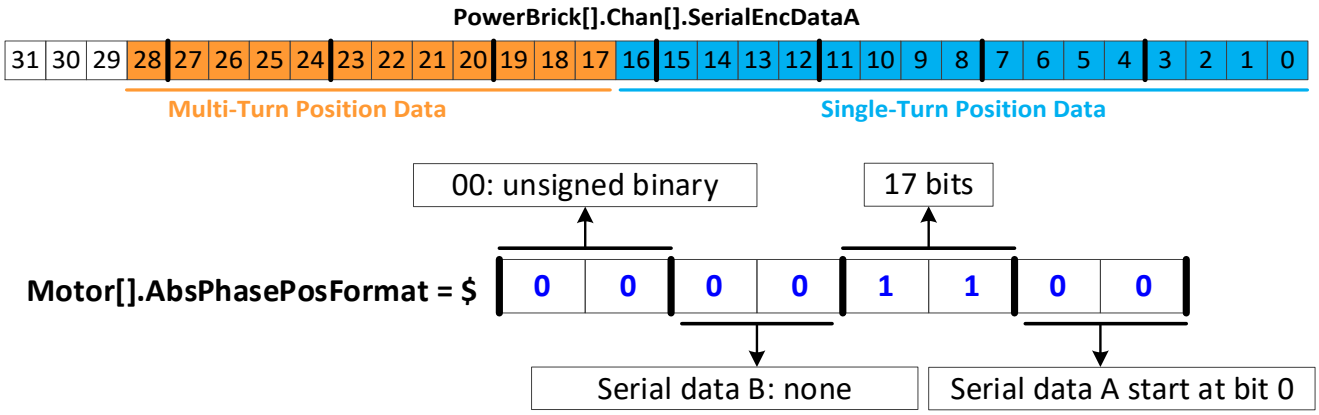
Motor[].pAbsPhasePos = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00002004
Rotary: Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{32}
Linear: Motor[].AbsPhasePosSf = 2048 * (**RES**$_{mm}$ * $2^{(36 - 32)}$) / **ECL**$_{mm}$
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = (PowerBrick [].Chan[].SerialEncDataB & $0000000F) << 28 +
(PowerBrick [].Chan[].SerialEncDataA & $FFFFFFF0) >> 4 L0


Note

Because this encoder is more than 32 bits, only the highest 32 bits are used. This requires alternate equations for **AbsPhasePosSf**.

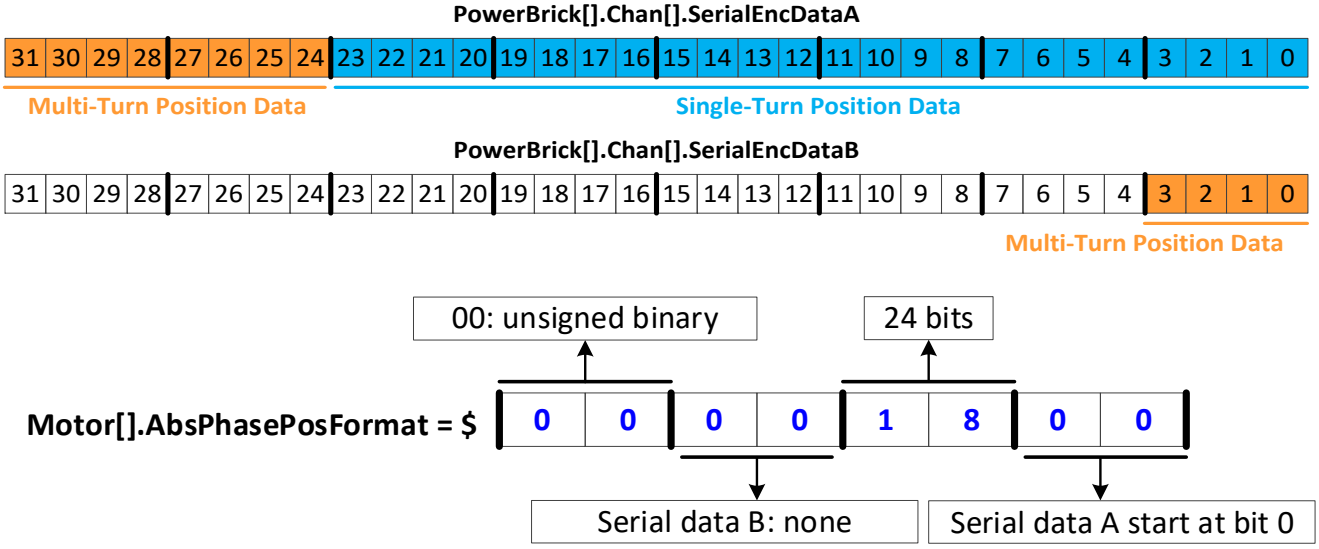
Example 4: A 29-bit binary serial encoder with 17 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA**.



Motor[].pAbsPhasePos = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00001100
Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{17}
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = PowerBrick[].Chan[].SerialEncDataA & $0001FFFF L0

Example 5: A 36-bit binary serial encoder with 24 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of **SerialEncDataA** and continuously extending to bit #3 of **SerialEncDataB**.



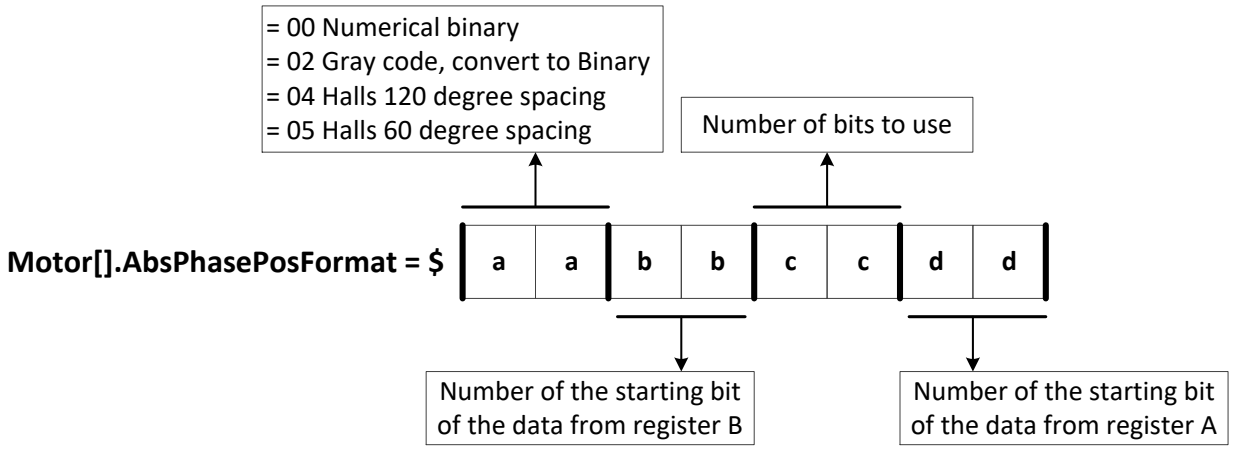
Motor[].pAbsPhasePos = PowerBrick[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00001800
Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{24}
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = PowerBrick[].Chan[].SerialEncDataA & $00FFFFFF L0

➢ **ABSOLUTE SERIAL ENCODER PHASING WITH ACC-84B**

With absolute serial encoders, the four key elements for setting up absolute phasing are:

- **Motor[].pAbsPhasePos = ACC84B[].Chan[].SerialEncDataA.a**
- **Motor[].AbsPhasePosFormat**
  - Encoders with no multi-turn position data are unsigned. Rotary encoders with multi-turn position data are signed.
  - Only 32 bits of position data can be used for absolute phasing. This should be the upper 32 bits of (single-turn) positon data.



- **Motor[].AbsPhasePosSf**
  If less than 32 **SingleTurnBits**
  Rotary: = 2048 * **NoOfPolesPairs** / $2^{\mathbf{SingleTurnBits}}$
  Linear: = 2048 * $\mathbf{RES_{mm}}$ / $\mathbf{ECL_{mm}}$
  If more than 32 **SingleTurnBits**
  Rotary: = 2048 * **NoOfPolesPairs** / 2^{32}
  Linear: = 2048 * $\mathbf{RES_{mm}}$ * $2^{\mathbf{SingleTurnBits-32}}$ / $\mathbf{ECL_{mm}}$

- **Motor[].AbsPhasePosOffset = –PhaseForceTest * Motor[].AbsPhasePosSf**
  Where:
  - **NoOfPolePairs** is the number of pole pairs of a rotary motor
  - **SingleTurnBits** is the number of bits of single turn position data for rotary serial encoder.
  - $\mathbf{ECL_{mm}}$ is the linear motor electrical cycle length or magnetic pitch (e.g. 60.96 mm)
  - $\mathbf{RES_{mm}}$ is the linear encoder resolution in the same unit as the ECL (e.g. 1 µm = 0.001 mm)
  - **PhaseforceTest** is the position value recorded from the Stepper Phasing Force Test.

Note

Only 32 bits of position data can be used for absolute phasing.

Note

Gray code conversion should be omitted here if it had been already implemented in **ACC84B[].Chan[].SerialEncCmd**.

Although data may appear to start at bit 0 in the script environment, internally it is only 24 bits starting at bit 8. This means the starting bit number is 8 more than would be expected from viewing **Acc84B[].Chan[].SerialEncDataA** in the watch window or terminal.

Stepper Phasing Force Test

The following, are the basic steps for performing the stepper phasing force test, which is similar to manual motor phasing:

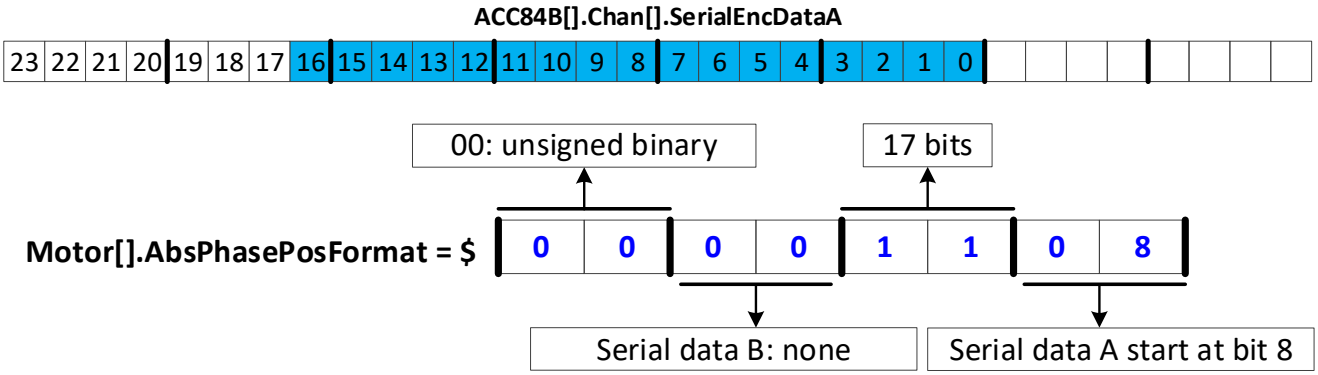
1. Make sure the motor is killed and steady.
2. Set **Motor[].IbBias** to a value corresponding to the amount of current to force into the phase. A conservative start is = **Motor[].I2TSet / 2.**
3. Issue a **#nOut0** (where n is the motor number). The motor should lock into a position and exhibit some stiffness when trying to move it by hand.

   Increase **Motor[].IbBias** as necessary until the motor is locked tightly. Exceeding the value of **Motor[].I2TSet** indicates that there is a problem with the amplifier output or that the motor or drive is not sized properly for the load.

   Wait for the motor to settle. In some instances, it may oscillate for an extended amount of time. Some motors may be small enough to safely stabilize by hand.

4. Record the entire position from Serial Data registers. See examples below for **PhaseForceTest** equations with masking and shifting.
5. Kill the motor; **#nK**.
6. Reset **Motor[].IbBias = 0**

Example 1: A binary serial encoder with 17 bits of single-turn (or an equivalent 1 µm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA**.



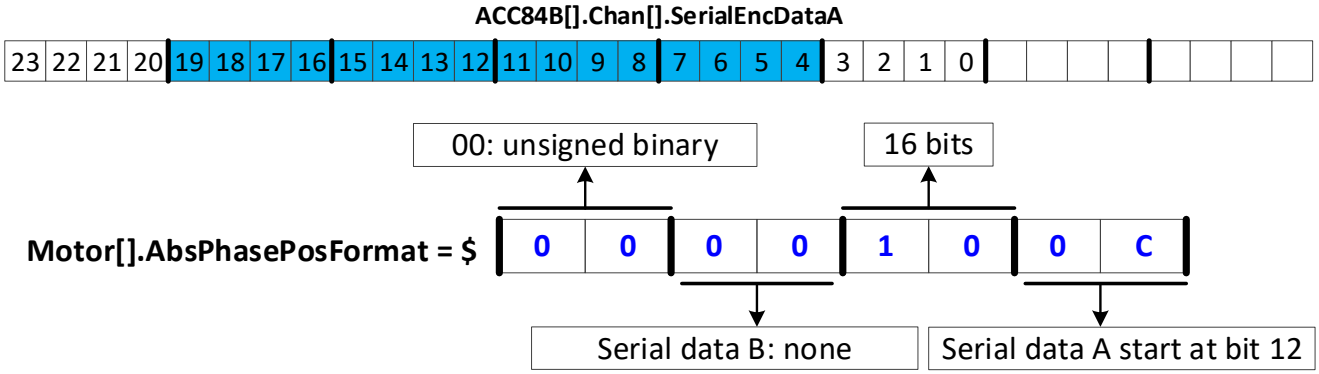
Motor[].pAbsPhasePos = ACC84B[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00001108
Rotary: Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{17}
Linear: Motor[].AbsPhasePosSf = 2048 * $\mathbf{RES_{mm}}$ / $\mathbf{ECL_{mm}}$
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = ACC84B[].Chan[].SerialEncDataA & $0001FFFF L0

Note

Internally data starts 8 bits to the left of what is shown in the watch window or terminal.

Example 2: A binary serial encoder with 16 bits of single-turn (or an equivalent 50 nm linear scale) position data starting at bit #4 of the 24 bit **SerialEncDataA**. The low 4 bits may contain other information, irrelevant to position data.



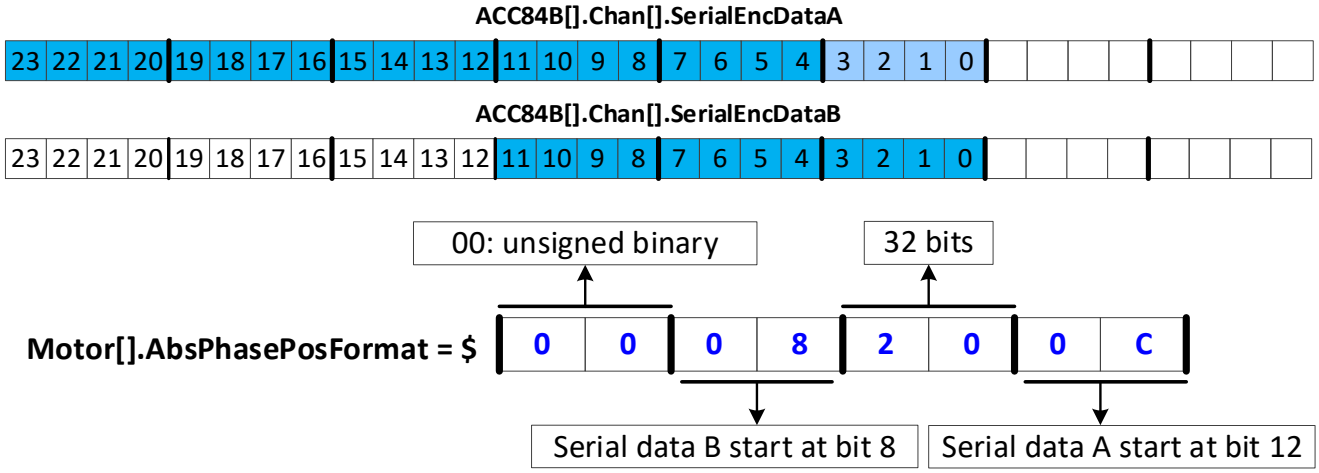
Motor[].pAbsPhasePos = ACC84B[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $0000100C
Rotary: Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{16}
Linear: Motor[].AbsPhasePosSf = 2048 * $\mathbf{RES_{mm}}$ / $\mathbf{ECL_{mm}}$
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = (ACC84B[].Chan[].SerialEncDataA & $000FFFF0) >> 4 L0

Note

Internally data starts 8 bits to the left of what is shown in the watch window or terminal.

Example 3: A binary serial encoder with 36 bits of single-turn (or an equivalent 1 nm linear scale) position data starting at bit #0 of the 24 bit **SerialEncDataA** and extending to bit #11 of **SerialEncDataB**. We will use the upper 32 bits; that is the maximum allowed number of bits for the power-on absolute commutation.



Motor[].pAbsPhasePos = ACC84B[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $0008200C
Rotary: Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{32}
Linear: Motor[].AbsPhasePosSf = 2048 * (**RES**$_{mm}$ * $2^{(36 - 32)}$) / **ECL**$_{mm}$
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.

```
L0 = (ACC84B[].Chan[].SerialEncDataB & $00000FFF) << 20 +
     (ACC84B[].Chan[].SerialEncDataA & $00FFFFF0) >> 4 L0
```

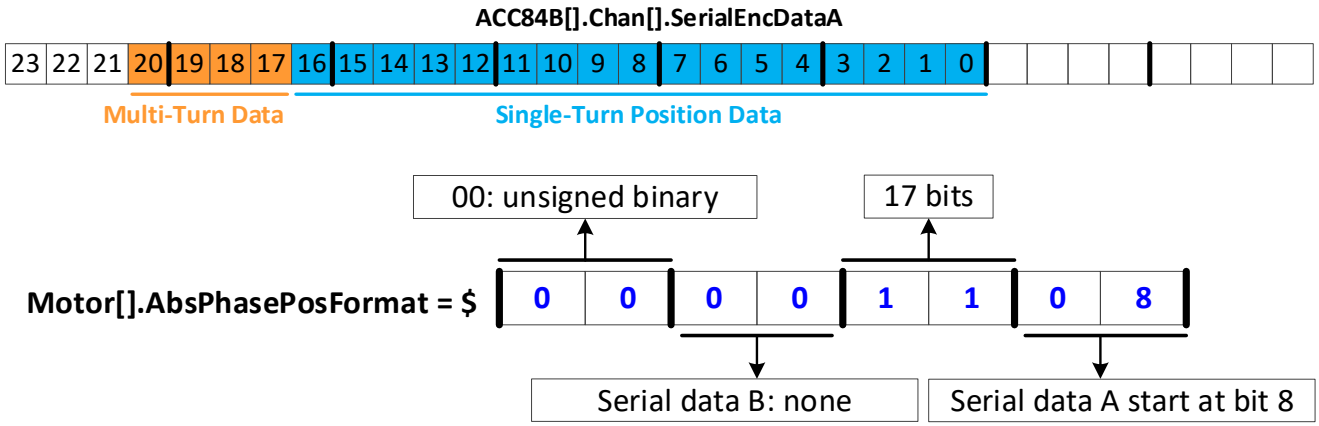
Note

Because this encoder is more than 32 bits, only the highest 32 bits are used. This requires alternate equations for **AbsPhasePosSf**.

Note

Internally data starts 8 bits to the left of what is shown in the watch window or terminal.

Example 4: A 21-bit binary serial encoder with 17 bits of single-turn and 4 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA**.



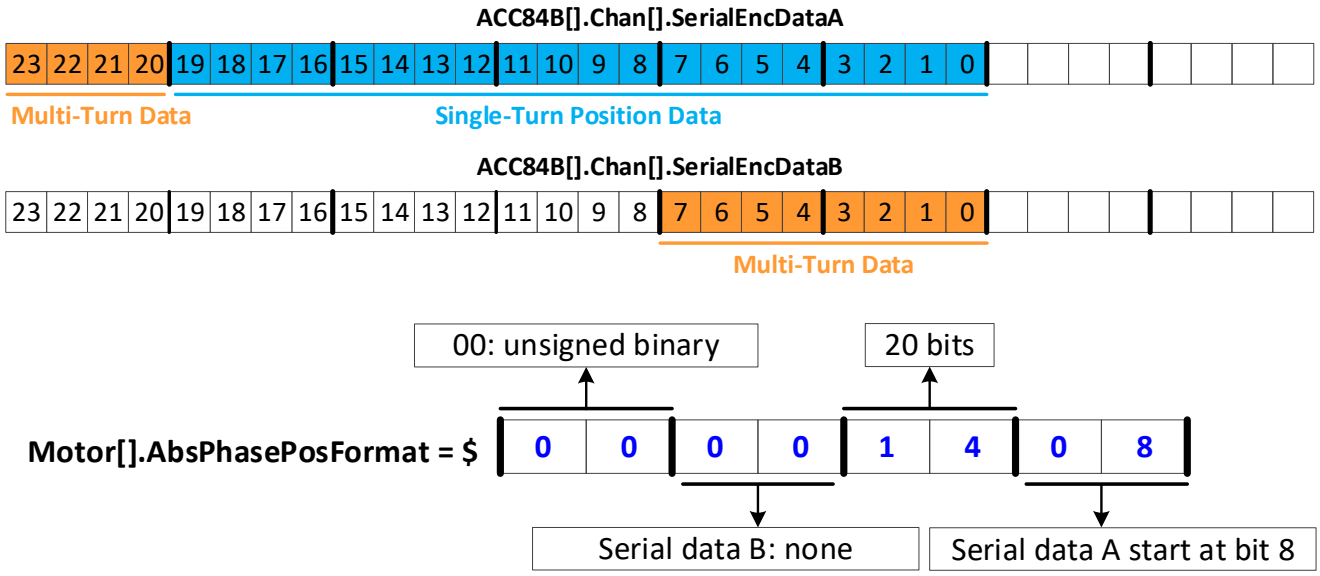
Motor[].pAbsPhasePos = ACC84B[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00001108
Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{17}
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = ACC84B[].Chan[].SerialEncDataA & $0000FFFF. **L0**

Note

Internally data starts 8 bits to the left of what is shown in the watch window or terminal.

Example 5: A 32-bit binary serial encoder with 20 bits of single-turn and 12 bits of multi-turn position data starting at bit #0 of the 24 bit **SerialEncDataA** and continuously extending to bit #7 of **SerialEncDataB**.



Motor[].pAbsPhasePos = ACC84B[].Chan[].SerialEncDataA.a
Motor[].AbsPhasePosFormat = $00001408
Motor[].AbsPhasePosSf = 2048 * **NoOfPolePairs** / 2^{20}
Motor[].AbsPhasePosOffset = –**PhaseForceTest** * Motor[].AbsPhasePosSf

The **PhaseForceTest** value can be found by performing a Stepper Phasing Force Test. The following command can be used to find the value at the correct step. The returned value should be used in the project.
L0 = ACC84B[].Chan[].SerialEncDataA & $000FFFFF L0

Note

Internally data starts 8 bits to the left of what is shown in the watch window or terminal.

➢ **ABSOLUTE RESOLVER PHASING**

With resolvers, the four key elements for setting up absolute phasing are:

- **Motor[].pAbsPhasePos = PowerBrick[].Chan[].AtanSumOfSqr.a**
- **Motor[].AbsPhasePosFormat = $00001010**
- **Motor[].AbsPhasePosSf**
  Rotary Motor: = **2048** * **NoOfPolesPairs** / **(ResPolePairs * 65536)**

  Where:
  - **NoOfPolePairs** is the number of pole pairs of a rotary motor
  - **ResPolePairs** is the resolver number of pole pairs.

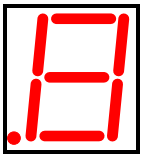
- **Motor[].AbsPhasePosOffset** = –**PhaseForceTest * Motor[].AbsPhasePosSf**
  Where: **PhaseforceTest** is the value recorded from the stepper phasing force test, by reading the upper 16 bits of **PowerBrick[].Chan[].AtanSumOfSqr.**

The **PhaseForceTest** value can be found by performing a manual force phasing (locking the motor onto phase B) and recording the value of the upper 16 bits of **AtanSumOfSrq** (e.g. **PhaseForceTest = PowerBrick[].Chan[].AtanSumOfSqr >> 16**).

SPECIAL FUNCTIONS & TROUBLESHOOTING

D1: Error Codes

The Power Brick LV utilizes a scrolling single-digit 7-segment display to exhibit amplifier faults. In normal operation mode (logic and DC bus power applied), the Power Brick LV will display a solid dot indicating that the software and hardware are running normally.



DISPLAY	DESCRIPTION
	Solid Dot: Normal mode operation. No fault (s)
GLOBAL FAULTS	
	Under Voltage: Indicates that the bus voltage is not present or less than 12 Volts
	Over Voltage: Indicates that the bus voltage has exceeded 85 Volts
	Over Temperature: Indicates that the (internal) electronics have exceeded 65°C
AXIS n FAULT (n = 1 through 8)	
	Axis n Over load: Indicates that channel n ‘s current rating (0.75 A / 3 A / 15 A) has been exceeded
	Axis n Over Current: Indicates that channel n’s peak current has exceeded the permissible limit (20 A)

Note

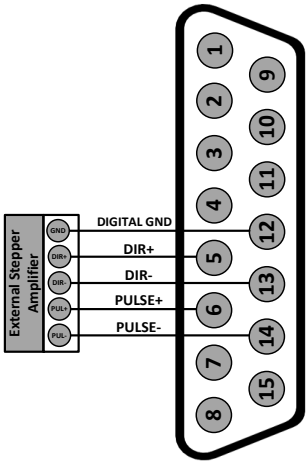
Clearing amplifier faults (and fault display) is done by enabling the power-on reset PLC or issuing a **BrickLV.Reset = 1** (requires waiting for pass/fail of operation).

Step and Direction, PFM Output

The Power Brick LV has the capability of generating step and direction output signals - aka PFM (Pulse Frequency Modulation) – for general purpose usage or control of external devices such as stepper amplifiers. The maximum pulse frequency and minimum pulse width are typically provided by the third party device manufacturer.

The step and direction outputs are RS422 compatible, +5 VDC level, and could be connected in either differential or single-ended configuration.

These PFM signals are generated out of the X1 – X8 encoder connectors, not to confuse them with the motor output connectors Amp1 – Amp8.



Pins #5, 6, 13, and 14 of the encoder feedback connectors (X1 – X8) share three different functions: only one of these functions (per channel) can be used – configured in software – at one time:

- Pulse and direction PFM output signals.
- TUVW hall flag inputs.
- Serial Encoder input.

Note

Each channel is independent of the other channels and can have its own use for these pins.

Most common usage of the PFM output signals:

- Manual modulation (from HMI or PLC, e.g. Laser modulation).
- Controlling an external stepper amplifier/motor.

➢ **COMMON CHANNEL SETTINGS FOR PFM OUTPUT (EXAMPLE CHANNEL 1):**

```
PowerBrick[0].Chan[0].PackOutData = 0                                    // Unpack Output Data
PowerBrick[0].Chan[0].OutputMode = PowerBrick[0].Chan[0].OutputMode | $8 // Force D PFM
PowerBrick[0].Chan[0].PfmFormat = 0                                      // =0 PFM, =1 Quadrature
PowerBrick[0].Chan[0].PfmDirPol = 0                                      // Non-Inverted
PowerBrick[0].Chan[0].OutFlagD = 1                                       // =0 for halls, =1 for PFM
```

➢ **PFM OUTPUT SIGNAL SETTINGS**

Next, we need to specify the frequency and pulse width. These parameters are defined by the two elements:

- **PowerBrick[].PfmClockDiv**
- **PowerBrick[].Chan[].PfmWidth**

The following PLC sample computes these elements automatically (e.g. first Gate, first channel) based on the user input of maximum PFM frequency [1.5 – 10000] KHz, and pulse width or duty cycle.

Note

The pulse width is specified in duty cycle if PfmDutyCycle [%] is non-zero, otherwise, it is computed in width [µsec] as entered in PulseWidth.

```
GLOBAL Fmax = 20            // USER INPUT [1.5 - 10000] KHz
GLOBAL PfmDutyCycle = 0     // USER INPUT [%]
GLOBAL PulseWidth = 25.6    // USER INPUT [µsec]

GLOBAL Fperiod
GLOBAL Fclk
GLOBAL MinPulseWidth
GLOBAL MaxPulseWidth
GLOBAL MinDutyCycle
GLOBAL MaxDutyCycle
GLOBAL CmdPulseWidth

OPEN PLC PfmCalcPLC
// Fmax CONSTRAINT [1.5 - 10000] KHz
IF(Fmax < 1.5){Fmax = 1.5}
IF(Fmax > 10000){Fmax = 10000}

// COMPUTE CURRENT
Fclk = 100000 / EXP2(Gate3[0].PfmClockDiv)
PowerBrick[0].PfmClockDiv = LOG2(100000 / Fclk)
Motor[1].MaxDac = 65536 * Fmax / Fclk

WHILE(Motor[1].MaxDac < 512)
{
   Fclk /= 2
   PowerBrick[0].PfmClockDiv = LOG2(100000 / Fclk)
   Motor[1].MaxDac = 65536 * Fmax / Fclk
}

WHILE(Motor[1].MaxDac > 32768)
{
   Fclk *= 2
   PowerBrick[0].PfmClockDiv = LOG2(100000 / Fclk)
   Motor[1].MaxDac = 65536 * Fmax / Fclk
}

// INCREASE ENCODER SAMPLING CLOCK?
IF(PowerBrick[0].PfmClockDiv < PowerBrick[0].EncClockDiv)
{
   PowerBrick[0].EncClockDiv = PowerBrick[0].PfmClockDiv
}
```

```
// COMPUTE MIN AND MAX
Fperiod = 1000 / Fmax
MinPulseWidth = 1000 / Fclk
MaxPulseWidth = INT((Fperiod - MinPulseWidth) / MinPulseWidth)  * MinPulseWidth
MinDutyCycle = MinPulseWidth * 100 / Fperiod
MaxDutyCycle = MaxPulseWidth * 100 / Fperiod

// COMMANDING USING DUTY CYCLE %?
IF(PfmDutyCycle > 0)
{
   // DUTY CYCLE CONSTRAINT
   IF(PfmDutyCycle > MaxDutyCycle)
   {
       PfmDutyCycle = MaxDutyCycle
   }

   IF(PfmDutyCycle < MinDutyCycle)
   {
       PfmDutyCycle = MinDutyCycle
   }
   PulseWidth = PfmDutyCycle * 1000 / (100 * Fmax)
}
// PULSE WIDTH CONSTRAINT
IF(PulseWidth > MaxPulseWidth)
{
   PulseWidth  = MaxPulseWidth
}

IF(PulseWidth < MinPulseWidth)
{
   PulseWidth  = MinPulseWidth
}

// COMPUTE PfmWidth SETTING
PowerBrick[0].Chan[0].PfmWidth = RINT(PulseWidth * Fclk / 1000)
CmdPulseWidth = PowerBrick[0].Chan[0].PfmWidth * MinPulseWidth

IF(CmdPulseWidth < PulseWidth)
{
   PowerBrick[0].Chan[0].PfmWidth += 1
   CmdPulseWidth = PowerBrick[0].Chan[0].PfmWidth * MinPulseWidth
}
DISABLE PLC PfmCalcPLC
CLOSE
```

Once executed, with the desired user input, this PLC produces

- **PowerBrick[].PfmClockDiv** (to be saved in the IDE project)
- **PowerBrick[].Chan[].PfmWidth** (to be saved in the IDE project)
- **Motor[].MaxDac** (to be used if setting up a motor)
- **CmdPulseWidth** (to be used in the PID gains if setting up a motor)

➢ **MANUAL MODULATION**

The PFM output can now be modulated manually by writing to the structure element **PowerBrick[].Chan[].Pfm,** as scaled below**.** For a 5 kHz output frequency:

```
PowerBrick[0].Chan[0].Pfm = 5 * 4294483.648 / Fmax
```

Note, that the associated motor # to this channel must be de-activated (Motor[].ServoCtrl = 0) in this mode.

➢ CONTROLLING AN EXTERNAL STEPPER AMPLIFIER / MOTOR

Closing the loop or jogging a motor driven by an external amplifier requires the following settings.

➢ EXAMPLE

```
PowerBrick[0].Chan[0].EncCtrl = 8                    // Internal Pulse And Direction
PowerBrick[0].Chan[0].TimerMode = 3                  // Read as PFM when looped back in with EncCtrl = 8

Motor[1].ServoCtrl = 1                               // Activate channel
Motor[1].pLimits = 0                                 // Disable overtravel limits?
Motor[1].pAmpFault = 0                               // Disable amplifier fault?
Motor[1].pDac = PowerBrick[0].Chan[0].Pfm.a          // Command output, point to PFM

Motor[1].MaxDac = 65536 * Fmax / Fclk

Motor[1].Servo.Kp = 2 * CmdPulseWidth * Fmax / 1000
Motor[1].Servo.Kvfb = 0
Motor[1].Servo.Kvifb = 0
Motor[1].Servo.Kvff = Motor[1].Servo.Kp * 5
Motor[1].Servo.Kviff = 0
Motor[1].Servo.Ki = Motor[1].Servo.Kp / 100
Motor[1].Servo.Kaff = 0
Motor[1].Servo.Kfff = 0

Motor[1].Servo.BreakPosErr = 1                       // Deadband size [motor units]
Motor[1].Servo.Kbreak = 0                            // Deadband gain

Motor[1].FatalFeLimit = 0                            // Fatal following limit
Motor[1].WarnFeLimit = 0                             // Fatal following warning
```

Motor[].MaxDac is the maximum command output which produces the maximum PFM speed (frequency) with a 100% open loop output. Do not issue this command with the signals connected to an amplifier/motor/device.

The servo PID gains can be tuned experimentally. And since this is a synthetic loop, the goal is to have sufficient gains to allow the maximum desired speed, and no pulsing at stand still. Adding a small Deadband can be helpful alleviating small pulses at zero velocity (due to quantization) when higher PFM frequencies are used.

Note

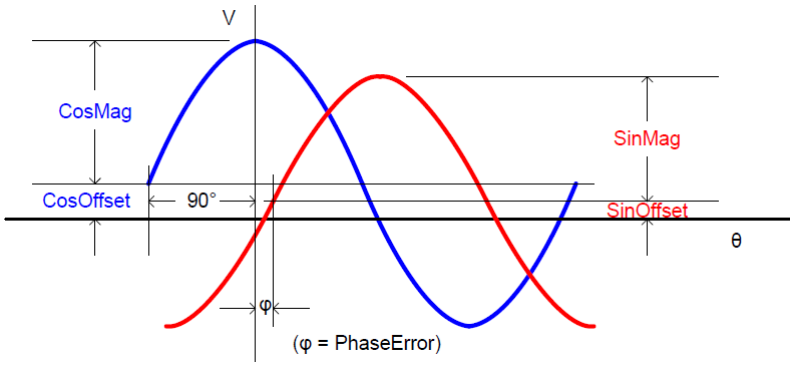
The maximum value(s) which **Motor[].JogSpeed** or **Motor[].MaxSpeed** can/should be set to is the previously computed **Fmax**.

Note

It is impossible to generate a PFM frequency higher than the encoder sampling rate in this mode. The encoder clock frequency **PowerBrick[].EncClockDiv** may need to be increased from the default of 3.125 MHz if higher frequencies are required.

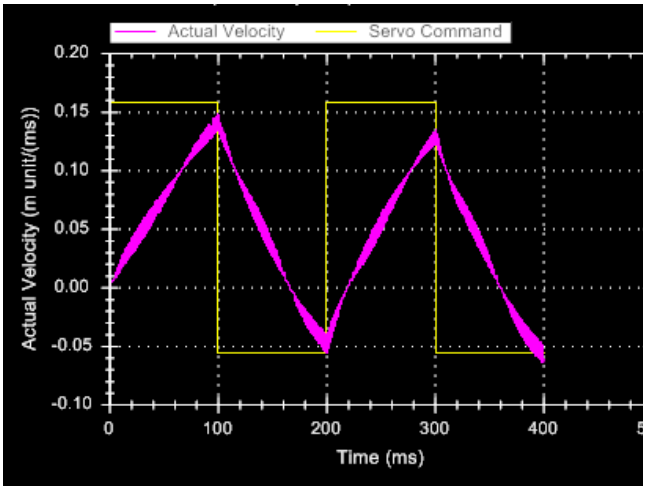
Sinusoidal Encoder Bias Corrections

Before computing the sub-count interpolated position with the arctangent calculation, the PMAC3-style ASIC can add in offset terms to the measured values in the ADC registers to compensate for voltage biases in the encoder and/or receiving circuitry.

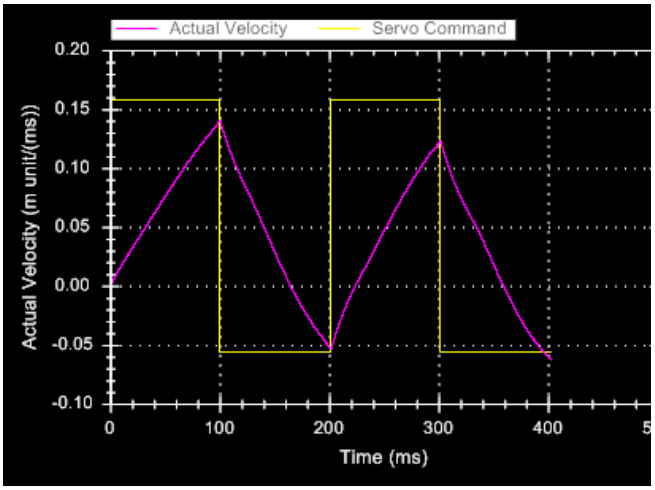


Excessive biases due to scale/read head misalignment, noise, or mechanical assembly may result in rough motion, visible velocity harmonics, and or encoder count loss. This is best seen with an open loop test:

Uncorrected



Corrected



Note

Corrections with the ACI (Auto Correcting Interpolator) are done automatically for both offsets and phase. This procedure is only suitable when interpolating with the DSPGate3 (x16384) – without the ACI option.

The Sine and Cosine offset structure elements are **PowerBrick[].Chan[].AdcOffset[0]** and **PowerBrick[].Chan[].AdcOffset[1]**, respectively.

Compensating for these offsets is done by reading the Sine – **PowerBrick[].Chan[].AdcEnc[0] –** and Cosine – **PowerBrick[].Chan[].AdcEnc[1] –** signals while moving the motor (preferably slowly) in open loop – by hand – or closed loop along the full travel, and computing their min and max values. The offset for each signal is then computed by averaging the min and the max values. The opposite value of this average is written into the corresponding offset register.

The following PLC program can assist in finding these bias offsets.

```
PTR Enc1Sine->PowerBrick[0].Chan[0].AdcEnc[0]
PTR Enc1Cosine->PowerBrick[0].Chan[0].AdcEnc[1]
GLOBAL Enc1SineOffset = 0, Enc1CosineOffset = 0

PTR Enc2Sine->PowerBrick[0].Chan[1].AdcEnc[0]
PTR Enc2Cosine->PowerBrick[0].Chan[1].AdcEnc[1]
GLOBAL Enc2SineOffset = 0, Enc2CosineOffset = 0

PTR Enc3Sine->PowerBrick[0].Chan[2].AdcEnc[0]
PTR Enc3Cosine->PowerBrick[0].Chan[2].AdcEnc[1]
GLOBAL Enc3SineOffset = 0, Enc3CosineOffset = 0

PTR Enc4Sine->PowerBrick[0].Chan[3].AdcEnc[0]
PTR Enc4Cosine->PowerBrick[0].Chan[3].AdcEnc[1]
GLOBAL Enc4SineOffset = 0, Enc4CosineOffset = 0

PTR Enc5Sine->PowerBrick[1].Chan[0].AdcEnc[0]
PTR Enc5Cosine->PowerBrick[1].Chan[0].AdcEnc[1]
GLOBAL Enc5SineOffset = 0, Enc5CosineOffset = 0

PTR Enc6Sine->PowerBrick[1].Chan[1].AdcEnc[0]
PTR Enc6Cosine->PowerBrick[1].Chan[1].AdcEnc[1]
GLOBAL Enc6SineOffset = 0, Enc6CosineOffset = 0

PTR Enc7Sine->PowerBrick[1].Chan[2].AdcEnc[0]
PTR Enc7Cosine->PowerBrick[1].Chan[2].AdcEnc[1]
GLOBAL Enc7SineOffset = 0, Enc7CosineOffset = 0

PTR Enc8Sine->PowerBrick[1].Chan[3].AdcEnc[0]
PTR Enc8Cosine->PowerBrick[1].Chan[3].AdcEnc[1]
GLOBAL Enc8SineOffset = 0, Enc8CosineOffset = 0

OPEN PLC SineCalPLC
LOCAL SineCycles = 0
LOCAL MaxEnc1Sine, MaxEnc1Cosine, MinEnc1Sine, MinEnc1Cosine
LOCAL MaxEnc2Sine, MaxEnc2Cosine, MinEnc2Sine, MinEnc2Cosine
LOCAL MaxEnc3Sine, MaxEnc3Cosine, MinEnc3Sine, MinEnc3Cosine
LOCAL MaxEnc4Sine, MaxEnc4Cosine, MinEnc4Sine, MinEnc4Cosine
LOCAL MaxEnc5Sine, MaxEnc5Cosine, MinEnc5Sine, MinEnc5Cosine
LOCAL MaxEnc6Sine, MaxEnc6Cosine, MinEnc6Sine, MinEnc6Cosine
LOCAL MaxEnc7Sine, MaxEnc7Cosine, MinEnc7Sine, MinEnc7Cosine
LOCAL MaxEnc8Sine, MaxEnc8Cosine, MinEnc8Sine, MinEnc8Cosine
```

```
WHILE(1)
{
   // ================== ENCODER 1 BIAS CORRECTIONS ================== //
   IF (SineCycles == 0)
   {
       MaxEnc1Sine = Enc1Sine
       MinEnc1Sine = Enc1Sine
       MaxEnc1Cosine = Enc1Cosine
       MinEnc1Cosine = Enc1Cosine
   }
   IF (Enc1Sine > MaxEnc1Sine){MaxEnc1Sine = Enc1Sine}
   IF (Enc1Sine < MinEnc1Sine){MinEnc1Sine = Enc1Sine}
   IF (Enc1Cosine > MaxEnc1Cosine){MaxEnc1Cosine = Enc1Cosine}
   IF (Enc1Cosine < MinEnc1Cosine){MinEnc1Cosine = Enc1Cosine}
   Enc1SineOffset = - (MaxEnc1Sine + MinEnc1Sine) / (2 * 65536)
   Enc1CosineOffset = - (MaxEnc1Cosine + MinEnc1Cosine) / (2 * 65536)
   // ================================================================ //

   // ================== ENCODER 2 BIAS CORRECTIONS ================== //
   IF (SineCycles == 0)
   {
       MaxEnc2Sine = Enc2Sine
       MinEnc2Sine = Enc2Sine
       MaxEnc2Cosine = Enc2Cosine
       MinEnc2Cosine = Enc2Cosine
   }
   IF (Enc2Sine > MaxEnc2Sine){MaxEnc2Sine = Enc2Sine}
   IF (Enc2Sine < MinEnc2Sine){MinEnc2Sine = Enc2Sine}
   IF (Enc2Cosine > MaxEnc2Cosine){MaxEnc2Cosine = Enc2Cosine}
   IF (Enc2Cosine < MinEnc2Cosine){MinEnc2Cosine = Enc2Cosine}
   Enc2SineOffset = - (MaxEnc2Sine + MinEnc2Sine) / (2 * 65536)
   Enc2CosineOffset = - (MaxEnc2Cosine + MinEnc2Cosine) / (2 * 65536)
   // ================================================================ //

   // ================== ENCODER 3 BIAS CORRECTIONS ================== //
   IF (SineCycles == 0)
   {
       MaxEnc3Sine = Enc3Sine
       MinEnc3Sine = Enc3Sine
       MaxEnc3Cosine = Enc3Cosine
       MinEnc3Cosine = Enc3Cosine
   }
   IF (Enc3Sine > MaxEnc3Sine){MaxEnc3Sine = Enc3Sine}
   IF (Enc3Sine < MinEnc3Sine){MinEnc3Sine = Enc3Sine}
   IF (Enc3Cosine > MaxEnc3Cosine){MaxEnc3Cosine = Enc3Cosine}
   IF (Enc3Cosine < MinEnc3Cosine){MinEnc3Cosine = Enc3Cosine}
   Enc3SineOffset = - (MaxEnc3Sine + MinEnc3Sine) / (2 * 65536)
   Enc3CosineOffset = - (MaxEnc3Cosine + MinEnc3Cosine) / (2 * 65536)
   // ================================================================ //
```

```
// ================== ENCODER 4 BIAS CORRECTIONS ================== //
IF (SineCycles == 0)
{
    MaxEnc4Sine = Enc4Sine
    MinEnc4Sine = Enc4Sine
    MaxEnc4Cosine = Enc4Cosine
    MinEnc4Cosine = Enc4Cosine
}
IF (Enc4Sine > MaxEnc4Sine){MaxEnc4Sine = Enc4Sine}
IF (Enc4Sine < MinEnc4Sine){MinEnc4Sine = Enc4Sine}
IF (Enc4Cosine > MaxEnc4Cosine){MaxEnc4Cosine = Enc4Cosine}
IF (Enc4Cosine < MinEnc4Cosine){MinEnc4Cosine = Enc4Cosine}
Enc4SineOffset = - (MaxEnc4Sine + MinEnc4Sine) / (2 * 65536)
Enc4CosineOffset = - (MaxEnc4Cosine + MinEnc4Cosine) / (2 * 65536)
// ================================================================ //

// ================== ENCODER 5 BIAS CORRECTIONS ================== //
IF (SineCycles == 0)
{
    MaxEnc5Sine = Enc5Sine
    MinEnc5Sine = Enc5Sine
    MaxEnc5Cosine = Enc5Cosine
    MinEnc5Cosine = Enc5Cosine
}
IF (Enc5Sine > MaxEnc5Sine){MaxEnc5Sine = Enc5Sine}
IF (Enc5Sine < MinEnc5Sine){MinEnc5Sine = Enc5Sine}
IF (Enc5Cosine > MaxEnc5Cosine){MaxEnc5Cosine = Enc5Cosine}
IF (Enc5Cosine < MinEnc5Cosine){MinEnc5Cosine = Enc5Cosine}
Enc5SineOffset = - (MaxEnc5Sine + MinEnc5Sine) / (2 * 65536)
Enc5CosineOffset = - (MaxEnc5Cosine + MinEnc5Cosine) / (2 * 65536)
// ================================================================ //

// ================== ENCODER 6 BIAS CORRECTIONS ================== //
IF (SineCycles == 0)
{
    MaxEnc6Sine = Enc6Sine
    MinEnc6Sine = Enc6Sine
    MaxEnc6Cosine = Enc6Cosine
    MinEnc6Cosine = Enc6Cosine
}
IF (Enc6Sine > MaxEnc6Sine){MaxEnc6Sine = Enc6Sine}
IF (Enc6Sine < MinEnc6Sine){MinEnc6Sine = Enc6Sine}
IF (Enc6Cosine > MaxEnc6Cosine){MaxEnc6Cosine = Enc6Cosine}
IF (Enc6Cosine < MinEnc6Cosine){MinEnc6Cosine = Enc6Cosine}
Enc6SineOffset = - (MaxEnc6Sine + MinEnc6Sine) / (2 * 65536)
Enc6CosineOffset = - (MaxEnc6Cosine + MinEnc6Cosine) / (2 * 65536)
// ================================================================ //
```

```
    // ================== ENCODER 7 BIAS CORRECTIONS ================== //
    IF (SineCycles == 0)
    {
        MaxEnc7Sine = Enc7Sine
        MinEnc7Sine = Enc7Sine
        MaxEnc7Cosine = Enc7Cosine
        MinEnc7Cosine = Enc7Cosine
    }
    IF (Enc7Sine > MaxEnc7Sine){MaxEnc7Sine = Enc7Sine}
    IF (Enc7Sine < MinEnc7Sine){MinEnc7Sine = Enc7Sine}
    IF (Enc7Cosine > MaxEnc7Cosine){MaxEnc7Cosine = Enc7Cosine}
    IF (Enc7Cosine < MinEnc7Cosine){MinEnc7Cosine = Enc7Cosine}
    Enc7SineOffset = - (MaxEnc7Sine + MinEnc7Sine) / (2 * 65536)
    Enc7CosineOffset = - (MaxEnc7Cosine + MinEnc7Cosine) / (2 * 65536)
    // ================================================================ //

    // ================== ENCODER 8 BIAS CORRECTIONS ================== //
    IF (SineCycles == 0)
    {
        MaxEnc8Sine = Enc8Sine
        MinEnc8Sine = Enc8Sine
        MaxEnc8Cosine = Enc8Cosine
        MinEnc8Cosine = Enc8Cosine
    }
    IF (Enc8Sine > MaxEnc8Sine){MaxEnc8Sine = Enc8Sine}
    IF (Enc8Sine < MinEnc8Sine){MinEnc8Sine = Enc8Sine}
    IF (Enc8Cosine > MaxEnc8Cosine){MaxEnc8Cosine = Enc8Cosine}
    IF (Enc8Cosine < MinEnc8Cosine){MinEnc8Cosine = Enc8Cosine}
    Enc8SineOffset = - (MaxEnc8Sine + MinEnc8Sine) / (2 * 65536)
    Enc8CosineOffset = - (MaxEnc8Cosine + MinEnc8Cosine) / (2 * 65536)
    // ================================================================ //

    SineCycles++
}
CLOSE
```

Using this PLC, and implementing the Sine and Cosine offsets:

- Delete bias corrections for channels which are not of interest.
- Enable the PLC (**ENABLE PLC SineCalPLC**).
- This PLC does not actively write to any structure element(s) or move any motor(s).
- Insert **EncXSineOffset** and **EncXCosineOffset** (for a given channel X) in the watch window.
- Move the motor (preferably slowly) along the full travel, back and forth, jogging or by hand.
- The **EncXSineOffset** and **EncXCosineOffset** will stop changing when the maximum and minimum values are reached indicating that the offsets have been computed.
- Disable the PLC (**DISABLE PLC SineCalPLC**).
- Write the computed offsets into the corresponding channel's elements (upper 16 bits)

```
PowerBrick[0].Chan[0].AdcOffset[0] = Enc1SineOffset * 65536          // Channel 1 Sine Offset
PowerBrick[0].Chan[0].AdcOffset[1] = Enc1CosineOffset * 65536        // Channel 1 Cosine Offset

PowerBrick[0].Chan[1].AdcOffset[0] = Enc2SineOffset * 65536          // Channel 2 Sine Offset
PowerBrick[0].Chan[1].AdcOffset[1] = Enc2CosineOffset * 65536        // Channel 2 Cosine Offset

PowerBrick[0].Chan[2].AdcOffset[0] = Enc3SineOffset * 65536          // Channel 3 Sine Offset
PowerBrick[0].Chan[2].AdcOffset[1] = Enc3CosineOffset * 65536        // Channel 3 Cosine Offset

PowerBrick[0].Chan[3].AdcOffset[0] = Enc4SineOffset * 65536          // Channel 4 Sine Offset
PowerBrick[0].Chan[3].AdcOffset[1] = Enc4CosineOffset * 65536        // Channel 4 Cosine Offset

PowerBrick[1].Chan[0].AdcOffset[0] = Enc5SineOffset * 65536          // Channel 5 Sine Offset
PowerBrick[1].Chan[0].AdcOffset[1] = Enc5CosineOffset * 65536        // Channel 5 Cosine Offset

PowerBrick[1].Chan[1].AdcOffset[0] = Enc6SineOffset * 65536          // Channel 6 Sine Offset
PowerBrick[1].Chan[1].AdcOffset[1] = Enc6CosineOffset * 65536        // Channel 6 Cosine Offset

PowerBrick[1].Chan[2].AdcOffset[0] = Enc7SineOffset * 65536          // Channel 7 Sine Offset
PowerBrick[1].Chan[2].AdcOffset[1] = Enc7CosineOffset * 65536        // Channel 7 Cosine Offset

PowerBrick[1].Chan[3].AdcOffset[0] = Enc8SineOffset * 65536          // Channel 8 Sine Offset
PowerBrick[1].Chan[3].AdcOffset[1] = Enc8CosineOffset * 65536        // Channel 8 Cosine Offset
```

Note

This procedure is done once per installation. These offsets must be saved in the project file(s) / configuration for the life of the motor / encoder.

Note

The PLC should be discarded once the procedure is finished. It does not need to be saved in the application project. Variable names such as Enc1SineOffset can be replaced by numeric values.

Reversing Motor Jogging Direction

Choosing the direction sense may be difficult during the initial setup of a motor. It is usually much easier and less confusing to set the motor up in the wrong direction before correcting the direction sense.

Following, are the necessary steps with respect to each type of motor/encoder. All other settings can remain the same.

Care should be taken when changing values as some settings are often set in terms of others and it would be easy to accidentally add a double negative.

Caution

These settings should not be applied while the motor is energized. The motor must be killed before applying these changes.

Stepper without Encoder (Direct Microstepping)

- **Motor[].PhaseOffset =** – present value
- **Motor[].PwmSf =** – present value

Quadrature / Sinusoidal / Resolver

- **Motor[].PhaseOffset =** – present value
- **Motor[].PwmSf =** – present value
- **PowerBrick[].Chan[].EncCtrl =** the opposite decode of the present value (e.g. 7 or 3)

  If using resolver or halls absolute power-on phasing:
  - **Motor[].AbsPhasePosSf =** – present value
  - **Motor[].AbsPhasePosOffset = 2048** – present value

  If using resolver absolute power-on position:
  - **Motor[].AbsPosSf =** – present value

Incremental Serial Encoders

- **Motor[].PhaseOffset =** – present value
- **Motor[].PwmSf =** – present value
- **Motor[].PhasePosSf =** – present value
- **EncTable[].ScaleFactor =** – present value

Absolute Serial Encoders

- **Motor[].PhaseOffset =** – present value
- **Motor[].PwmSf =** – present value
- **Motor[].PhasePosSf =** – present value
- **EncTable[].ScaleFactor =** – present value
- **Motor[].AbsPosSf =** – present value
- **Motor[].AbsPhasePosSf =** – present value
- **Motor[].AbsPhasePosOffset = 2048** – present value

Note

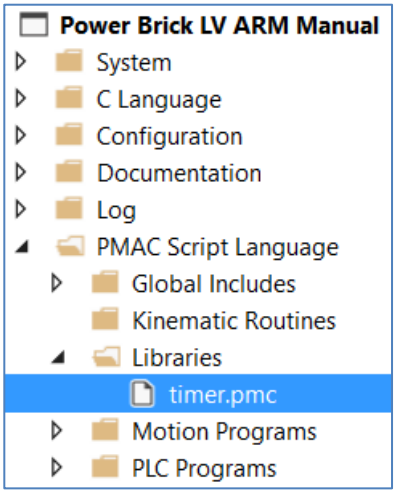
PMAC-commutated (e.g. Brushless) motors need to be phased again after applying these settings. All other settings (e.g. current & position loop tuning) should remain the same.

DelayTimer PLC

The following subprogram is a generic routine which is commonly called from PLC programs to insert a time delay in the logic process.

```
OPEN SUBPROG DelayTimer
SUB: sec (DelayTimeSec)
LOCAL EndTimeSec
EndTimeSec = Sys.Time + DelayTimeSec
WHILE (EndTimeSec > Sys.Time) {}
RETURN

SUB: msec (DelayTimeMsec)
LOCAL EndTimeMsec
EndTimeMsec = Sys.Time + DelayTimeMsec * 0.001
WHILE (EndTimeMsec > Sys.Time) {}
RETURN
CLOSE
```

This subprogram is automatically added to new projects in the “Libraries” folder of the Solution Explorer:



Calling **DelayTimer.sec** with a time argument specified in seconds or **DelayTimer.msec** with a time argument in milliseconds causes the desired delay in a script PLC, example:

```
GLOBAL MyToggle1 = 0
GLOBAL MyToggle2 = 0

OPEN PLC ExamplePLC
CALL DelayTimer.sec(1)      // 1 second time delay
MyToggle1 = MyToggle1 ^ 1   // Toggle variable1

CALL DelayTimer.msec(500)   // 500 millisecond time delay
MyToggle2 = MyToggle2 ^ 1   // Toggle variable2
CLOSE
```

Note

The Timer subprogram can be called from motion programs or other subprograms as well. However, for those types of programs the better suited and dedicated DWELL / DELAY commands are advised.

Encoder Count Error

The Power Brick LV is fitted with an encoder count error detection circuitry which supports **Quadrature**, **Sinusoidal**, **Resolver**, and **HiperFace** encoders.

The encoder count circuitry reports bad transitions of the quadrature signals. If both the A and B channels of the quadrature data change state at the decode circuitry (post-filter) in the same hardware sampling clock cycle, an unrecoverable error to the counter value will result (lost counts). **PowerBrick[].Chan[].CountError** is then set and latched to 1 (until reset or cleared). 0 indicates that there is no encoder count error.

```
PTR Enc1CountError->PowerBrick[0].Chan[0].CountError
PTR Enc2CountError->PowerBrick[0].Chan[1].CountError
PTR Enc3CountError->PowerBrick[0].Chan[2].CountError
PTR Enc4CountError->PowerBrick[0].Chan[3].CountError
PTR Enc5CountError->PowerBrick[1].Chan[0].CountError
PTR Enc6CountError->PowerBrick[1].Chan[1].CountError
PTR Enc7CountError->PowerBrick[1].Chan[2].CountError
PTR Enc8CountError->PowerBrick[1].Chan[3].CountError
```

Note

No automatic action is taken by the Power Brick LV if the encoder count error bit is set, it is the user's responsibility to trap it and create safety logic to stop the machine and / or alert the operator.

The encoder count error may not have immediate consequences on the motion, but it indicates ultimately that the motor is losing counts which could result in a fatal following error, erroneous commutation, or position drift over time.

Common root causes of the encoder count error:

- Encoder problem
- Trying to move the encoder (motor) faster than it's specification
- Using a higher resolution/speed encoder. This may require increasing the sampling clock.

The default sampling clock of ~3.125 MHz is acceptable for the majority of applications.

Increasing the encoder sampling clock is done using the structure element **PowerBrick[].EncClockDiv** (default = 5).

Setting	Frequency		Setting	Frequency	
0	100	MHz	8	390.6	kHz
1	50	MHz	9	195.3	kHz
2	25	MHz	10	97.65	kHz
3	12.5	MHz	11	48.82	kHz
4	6.25	MHz	12	24.41	kHz
5	**3.125**	**MHz**	13	12.21	kHz
6	1.562	MHz	14	3.104	kHz
7	781.2	kHz	15	3.052	kHz

Encoder Loss Detection

Warning

Loss of the feedback sensor signal is potentially a very dangerous condition in closed-loop control, because the servo loop no longer has any idea what the true physical position of the motor is – usually it thinks it is "stuck" – and it can react wildly, often causing a runaway condition.

The Power Brick LV has circuitry dedicated to monitoring the presence of a proper feedback signal. In addition, it can automatically check these circuits for loss of sensor signal and take appropriate shutdown action. This feature supports the following types of encoders:

- Digital quadrature (differential)
- Sinusoidal
- Resolver
- HiperFace
- Serial Encoders

Digital Quadrature

With digital quadrature encoders (must be differential) the encoder loss circuitry monitors each quadrature input pair with an exclusive-or XOR gate:

In normal operation mode, the two quadrature inputs should be in opposite logical states – that is one high and one low – yielding a true output from the XOR gate.

When there is no longer a proper signal driving the inputs on the interface, both lines are pulled to a high logical level internally, so the XOR gate outputs a low level indicating encoder loss.

The flag reflecting the encoder loss status is found in the Power Brick bit element **PowerBrick[].Chan[].LossStatus**:

- Test
- = 0 in normal mode.
- = 1 (and latched) upon detecting an encoder loss

```
PTR Enc1LossBit->PowerBrick[0].Chan[0].LossStatus
PTR Enc2LossBit->PowerBrick[0].Chan[1].LossStatus
PTR Enc3LossBit->PowerBrick[0].Chan[2].LossStatus
PTR Enc4LossBit->PowerBrick[0].Chan[3].LossStatus
PTR Enc5LossBit->PowerBrick[1].Chan[0].LossStatus
PTR Enc6LossBit->PowerBrick[1].Chan[1].LossStatus
PTR Enc7LossBit->PowerBrick[1].Chan[2].LossStatus
PTR Enc8LossBit->PowerBrick[1].Chan[3].LossStatus
```

Automatic Kill Action for Quadrature Encoders

Arming the automatic kill action with quadrature encoders:

```
Motor[1].pEncLoss = PowerBrick[0].Chan[0].Status.a
Motor[1].EncLossBit = 28
Motor[1].EncLossLevel = 1 // High true fault
Motor[1].EncLossLimit = 0
```

In this mode, the status of the encoder loss can be monitored in the motor status window in the IDE software or using the motor element bit **Motor[].EncLoss**.

Note

Setting **Motor[].pEncLoss** = 0 disables the automatic kill action.

Sinusoidal | Resolver | HiperFace Encoders

Analog sinusoidal encoders and resolvers provide simultaneous sine and cosine signals into the analog-to-digital converters of the Power Brick LV interface circuitry.

In proper operation, the sum of the squares of the converted values for these two signals should be roughly constant, and significantly different from zero.

The Power Brick LV ASIC computes this sum-of-squares value every sample cycle. The latest value is always available in the 16-bit element **PowerBrick[].Chan[].SumOfSquares**.
In addition, if all of the highest 4 bits of this element are zero, so the value is less than 1/16 of full range, the status bit **PowerBrick[].Chan[].SosError** is automatically set to 1.

```
PTR Enc1LossBit->PowerBrick[0].Chan[0].SosError
PTR Enc2LossBit->PowerBrick[0].Chan[1].SosError
PTR Enc3LossBit->PowerBrick[0].Chan[2].SosError
PTR Enc4LossBit->PowerBrick[0].Chan[3].SosError
PTR Enc5LossBit->PowerBrick[1].Chan[0].SosError
PTR Enc6LossBit->PowerBrick[1].Chan[1].SosError
PTR Enc7LossBit->PowerBrick[1].Chan[2].SosError
PTR Enc8LossBit->PowerBrick[1].Chan[3].SosError
```

Automatic Kill Action for Sinusoidal | Resolver | HiperFace Encoders

Arming the automatic kill action with Sinusoidal, Resolver, or HiperFace encoders:

```
Motor[1].pEncLoss = PowerBrick[0].Chan[0].SosError.a
Motor[1].EncLossBit = 31
Motor[1].EncLossLevel = 1 // High true fault
Motor[1].EncLossLimit = 5 // 5 scans
```

In this mode, the status of the encoder loss can be monitored in the motor status window in the IDE software or using the motor element bit **Motor[].EncLoss**.

Note

Setting **Motor[].pEncLoss** = 0 disables the automatic kill action.

Serial Encoders

The Power Brick LV provides interfaces for many of the most popular serial encoder protocols. For most of these interfaces, the receiving logic can detect that no data has been received in response to the cycle's "position request" output, and set a "timeout error" flag that can be read by the processor. This flag bit can be used to detect encoder loss.

This "timeout error" flag is bit 31 of the element **PowerBrick[].Chan[].SerialEncDataB**.

It is also possible to utilize an error-checking mechanism in the data such as parity or cyclic redundancy check (CRC) bits. The Power Brick LV can evaluate these mechanisms and determine whether the data set was valid or not. This is particularly recommended for the SSI protocol, where the data patterns cannot be used to detect a timeout error. For the SSI protocol, the parity error flag is bit 31 of **PowerBrick[].Chan[].SerialEncDataB**.

Automatic Kill Action for Gate3 Serial Encoders

Arming the automatic kill action for Gate3 serial encoder protocols with a "timeout error" flag (EnDat, HiperFace, Sigma I, Sigma II/III/V, Tamagawa, Panasonic, Mitutoyo, and Kawasaki):

```
Motor[1].pEncLoss = PowerBrick[0].Chan[0].SerialEncDataB.a
Motor[1].EncLossBit = 31  // Time out error bit number
Motor[1].EncLossLevel = 1 // High true fault
Motor[1].EncLossLimit = 0
```

Note

The same settings are valid for an SSI encoder with parity checking to use the parity-error bit.

Note

Setting **Motor[].pEncLoss** = 0 disables the automatic kill action.

Automatic Kill Action for Gate3 Serial Encoders

Arming the automatic kill action for ACC84B serial encoder protocols with a "timeout error" flag on **SerialEncDataB** (EnDat, Sigma II/III/V, BiSS, and Kawasaki):

```
Motor[1].pEncLoss = ACC84B[0].Chan[0].SerialEncDataB.a
Motor[1].EncLossBit = 31  // Time out error bit number
Motor[1].EncLossLevel = 1 // High true fault
Motor[1].EncLossLimit = 0
```

Note

The same settings are valid for an SSI encoder with parity checking to use the parity-error bit.

Arming the automatic kill action for ACC84B serial encoder protocols with a "timeout error" flag on **SerialEncDataC** (Tamagawa, Panasonic, Mitutoyo , Matsushita, and Mitsubishi):

```
Motor[1].pEncLoss = ACC84B[0].Chan[0].SerialEncDataC.a
Motor[1].EncLossBit = 31  // Time out error bit number
Motor[1].EncLossLevel = 1 // High true fault
Motor[1].EncLossLimit = 0
```

In this mode, the status of the encoder loss can be monitored in the motor status window in the IDE software or using the motor element bit **Motor[].EncLoss**.

Note

Setting **Motor[].pEncLoss** = 0 disables the automatic kill action.

Digital Tracking Filter

The encoder conversion table's (ECT) software tracking filter is a digital low-pass filter with an integrator which is useful for reducing measurement noise (floor level and occasionally electrical) without introducing steady-state error at constant velocity or position. It is particularly useful for applications involving:

- Analog Input Signals.
- Sinusoidal Encoder Signals (with the x16384 interpolator).
- Resolver Signals.

Note

Executed in the ECT, the performance of this filter is directly proportional to the servo frequency. The higher the frequency, the faster is the sampling and better noise rejection.

Note

This filter should never be used with the Sinusoidal ACI interpolation option of the Power Brick. The ACI automatically compensates for disturbances at a much higher rate.

The following PLC depicts the digital tracking filter equations, and produces the three indexes necessary to apply the filter in the corresponding Encoder Conversion Table:

```
GLOBAL TrackFltrCutOff = 1000       // [Hz]
GLOBAL TrackFltrDamping = 1
GLOBAL TrackFltrIndex1, TrackFltrIndex2, TrackFltrIndex4
GLOBAL VerifyWn, VerifyTau

OPEN PLC TrackFltrPLC
LOCAL TrackFltrTime = Sys.Servoperiod / 1000

TrackFltrIndex1 = 65
TrackFltrIndex2 = INT(256 - 512 * TrackFltrCutOff * TrackFltrDamping * TrackFltrTime)
TrackFltrIndex4 = 1

VerifyWn = (1 / TrackFltrTime) * SQRT(TrackFltrIndex1 / (256 * EXP2(TrackFltrIndex4)))
VerifyTau = (256 - TrackFltrIndex2) / (2 * SQRT(256 * TrackFltrIndex1 / EXP2(TrackFltrIndex4)))

WHILE (VerifyWn > TrackFltrCutOff)
{
   TrackFltrIndex4 += 1
   VerifyWn = (1 / TrackFltrTime) * SQRT(TrackFltrIndex1 / (256 * EXP2(TrackFltrIndex4)))
}

WHILE (VerifyWn < TrackFltrCutOff)
{
   TrackFltrIndex1 += 1
   VerifyWn = (1 / TrackFltrTime) * SQRT(TrackFltrIndex1 / (256 * EXP2(TrackFltrIndex4)))
}

VerifyWn = (1 / TrackFltrTime) * SQRT(TrackFltrIndex1 / (256 * EXP2(TrackFltrIndex4)))
VerifyTau = (256 - TrackFltrIndex2) / (2 * SQRT(256 * TrackFltrIndex1 / EXP2(TrackFltrIndex4)))

DISABLE PLC TrackFltrPLC
CLOSE
```

This PLC example is very simple to use:

- Specify the desired cutoff frequency (~30 – 2000 Hz)
- Specify the desired damping ratio (typically 1.0)

For example, for a 1,000 Hz cutoff frequency, and 1 damping ratio will produce:

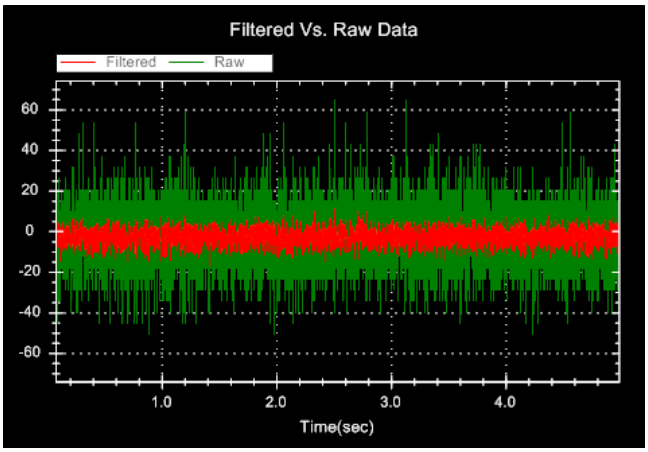
```
TrackFltrCutOff = 1000 TrackFltrDamping = 1 ENABLE PLC TrackFltrPLC


TrackFltrIndex1,3
P8203=82
P8204=154
P8205=3
```

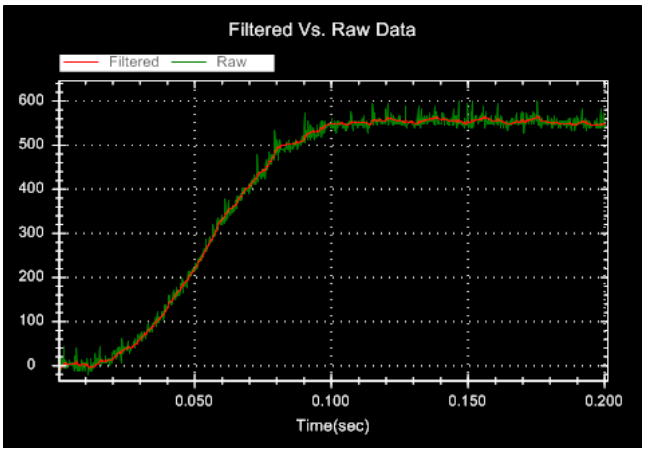
These index values copied into **EncTable[].index1**, **EncTable[].index2**, and **EncTable[].index4** respectively will apply the desired filtering.

Note

If these indexes are non-zero in the ECT entry of interest, another entry needs to be created, with its source pointing to the original entry's result **EncTable[].PrevEnc.a**.

The following plots show an example of a filtered "noisy" signal at steady state and in dynamic motion:





PTC Motor Thermal Input

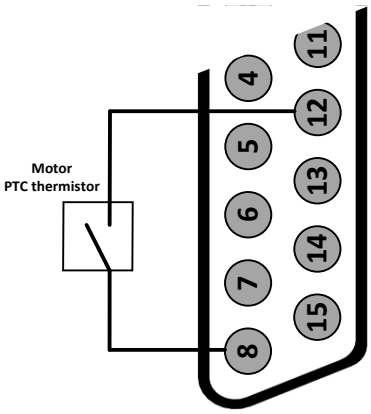
The PTC motor thermal Input (pin #8 of X1 – X8 connectors) is typically used to bring in motor over-temperature thermistor signal(s) into the Power Brick LV. Proper action can then be taken to safely stop operation if the motor is overheated.

Caution

No automatic action is taken by the Power Brick LV when the PTC input is triggered, it is the user's responsibility to trap it (i.e. in a background PLC) and insert safety logic to stop the motor and / or alert the operator.

The PTC input (pin #8) is typically wired to ground (pin #12) in series with the motor PTC thermistor:

- In normal mode operation, the circuit is open and PTC (pin #8) is pulled up to +5 VDC internally, this corresponds to a setting of 1 in software.
- If the motor is overheated, the circuit is closed and PTC (pin #8) is pulled down to ground (pin #12), this corresponds to a setting of 0 in software.




```
PTR Ch1PTC->PowerBrick[0].GpioData[0].24.1      // Channel 1 PTC Input, X1
PTR Ch2PTC->PowerBrick[0].GpioData[0].25.1      // Channel 2 PTC Input, X2
PTR Ch3PTC->PowerBrick[0].GpioData[0].26.1      // Channel 3 PTC Input, X3
PTR Ch4PTC->PowerBrick[0].GpioData[0].27.1      // Channel 4 PTC Input, X4

PTR Ch5PTC->PowerBrick[1].GpioData[0].24.1      // Channel 5 PTC Input, X5
PTR Ch6PTC->PowerBrick[1].GpioData[0].25.1      // Channel 6 PTC Input, X6
PTR Ch7PTC->PowerBrick[1].GpioData[0].26.1      // Channel 7 PTC Input, X7
PTR Ch8PTC->PowerBrick[1].GpioData[0].27.1      // Channel 8 PTC Input, X8
```

Note

If the PTC function is not used, this input can be used as general purpose relay input.

LED Status

Symbol	Description	Status		Indication
ABORT INPUT STATUS	Abort Status	Green	●	Enabled and 24V wired in
		Red	●	Disabled / Enabled and 24V not wired in
MACRO LINK	MACRO	Green	●	MACRO connected / Operational
		Red	●	MACRO not connected / Ring broken
DIAG.	USB Mode	Green	●	USB Mass Storage
		Amber	●	Serial Communications
RDY	Ready	Green	●	PMAC ready, boot complete
PWR/WD	Power/Watchdog	Green	●	Logic Power Connected
		Red	●	Fault – Watchdog

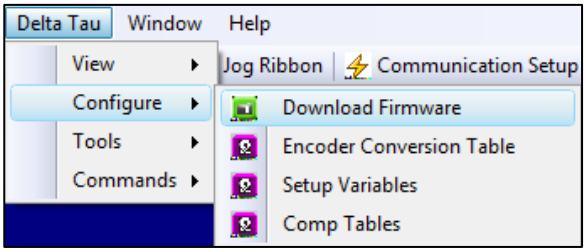
Reloading Power PMAC Firmware

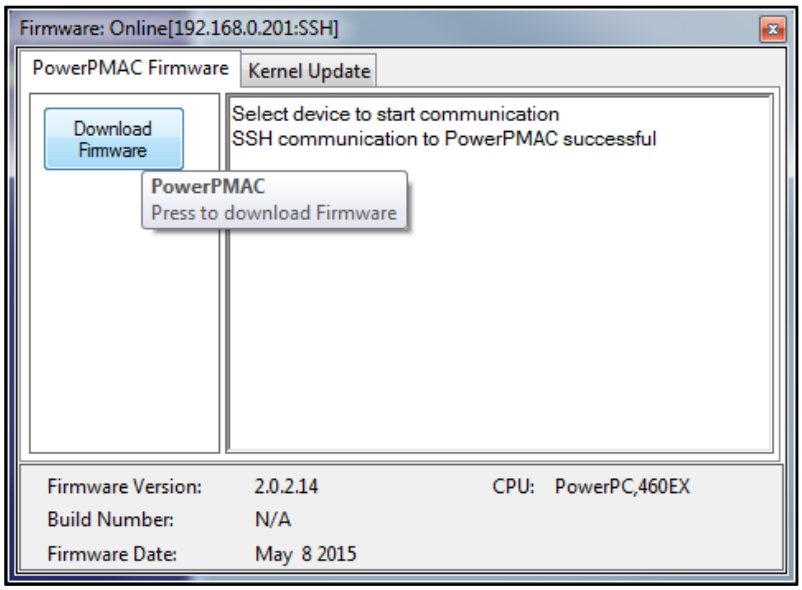
You should always use the newest released version of the Power PMAC firmware if your application permits.

Power PMAC Firmware can be reloaded by means of the IDE or a USB flash drive/SD card.

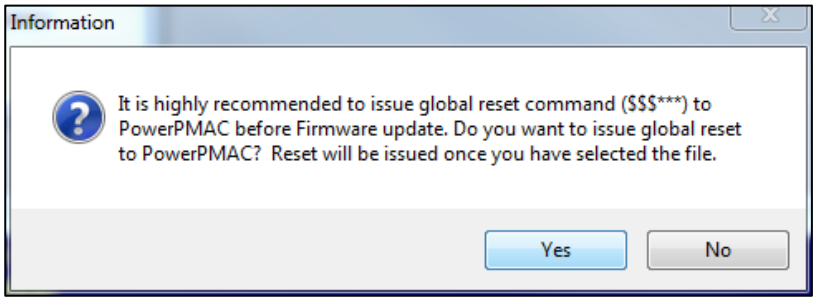
Reloading Firmware Method 1: IDE

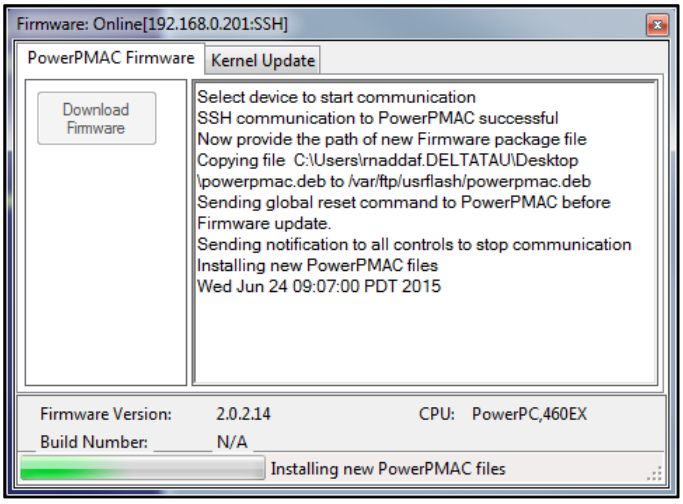
To install the latest firmware through the IDE, click on Delta Tau→Configure→Download Firmware:



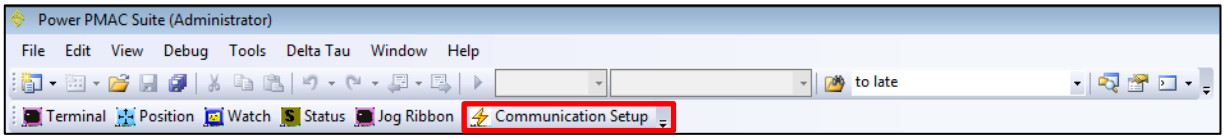


Under the "PowerPMAC Firmware" tab, click the "Download Firmware" button. On clicking the button, the IDE will ask whether it is acceptable to issue $$$*** (Global Reset) before updating the firmware. If you click "Yes," then the Power PMAC will be reset to factory default. It will then prompt you to browse for the firmware file you want to download. It is recommended to issue $$$*** and make sure that the Power PMAC is at factory default stage before downloading firmware, because the firmware download process does not kill the C background programs that are already executing. In this case, the firmware update process may not update all the files.

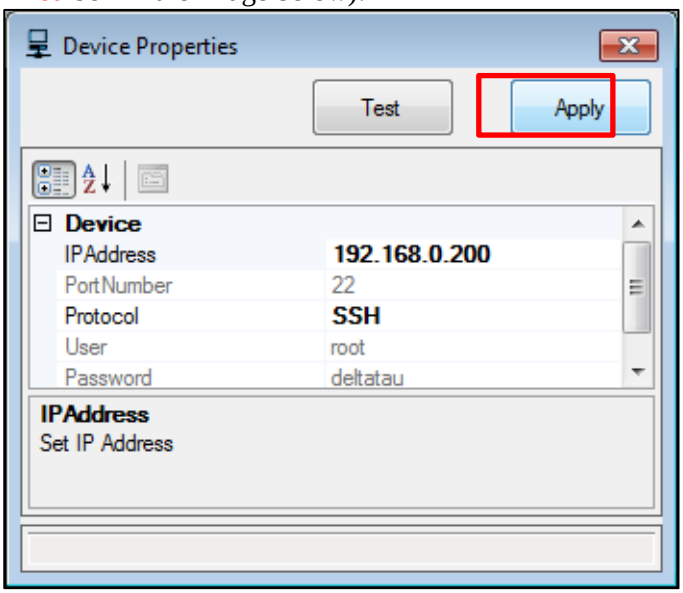




Wait for the IDE to finish downloading the firmware file and then for Power PMAC to reboot. If it does not reconnect successfully after rebooting, click "Communication Setup" (see the red box in the image below):



Then, click "Apply" (see the red box in the image below):



If you still cannot communicate, cycle power on the UMAC rack and use the "Communication Setup" button in the IDE again until you can connect.

Reloading Firmware Method 2: USB Drive/SD Card

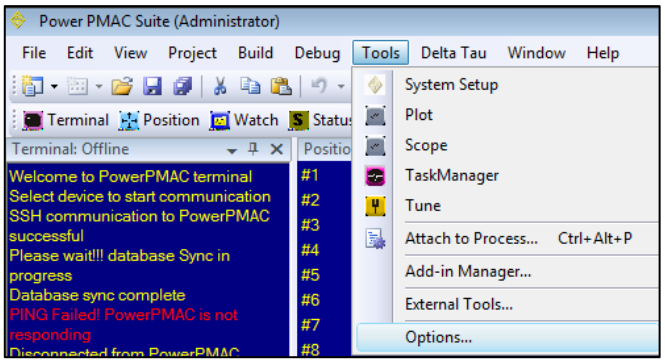
- Connect a USB memory stick/SD Card to a PC using any OS which can work with FAT32 partition.
- Create a folder named **PowerPmacFirmwareInstall** on the USB memory stick/SD Card root folder. Place the installation package "powerpmac.deb" into this folder.
- Safely remove the USB memory stick/SD Card from the PC.
- Plug the USB memory stick/SD Card into Power PMAC's USB port.

- If Power Brick is off, then power up the device. If Power Brick is on, cycle power.
- After the unit boots, the new firmware should be installed.

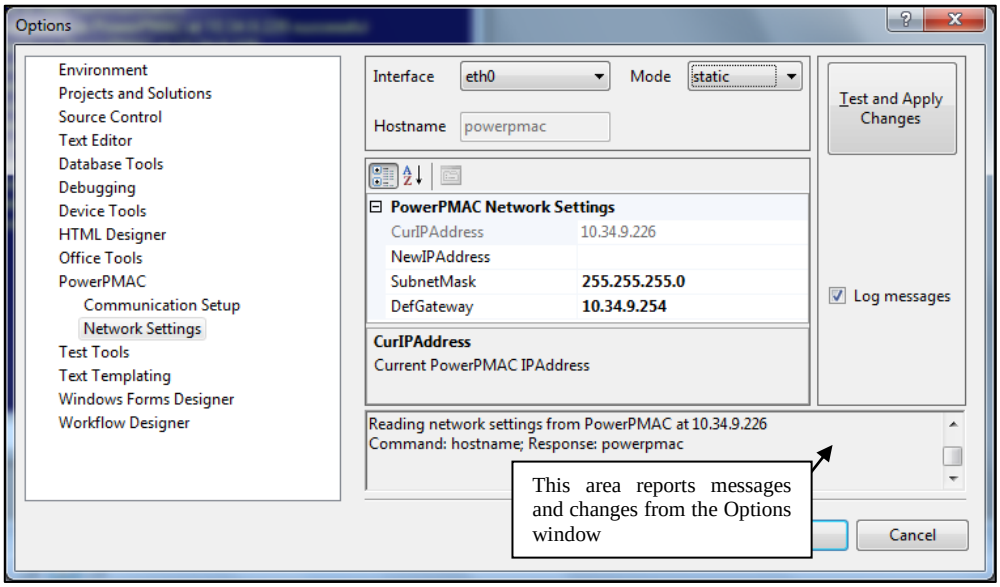
Changing Network (IP Address) Settings

Through the Power PMAC IDE

If you want to change Power PMAC's IP Address from within the IDE, click Tools→Options…



Near the bottom of the screen in the left pane, click PowerPMAC→Network Settings and then the following window should appear:



On this window, enter the **NewIPAddress** desired. You can enter this IP Address into the **DefGateway** field also. Leave the **SubnetMask** as 255.255.255.0. Then, click Test and Apply Changes.

Through USB

Below is the procedure for using a USB flash drive to detect/change the Power PMAC IP address:

1. Connect the USB memory stick/SD Card to your PC using any OS which can work with FAT32 partition.
2. Create a folder named **PowerPmacIP** on the USB memory stick/SD Card root folder.
3. Safely remove the USB memory stick/SD Card from the PC.
4. Plug the USB memory stick/SD Card into Power PMAC's USB port.
5. If Power Brick is off, then power up the device. If Power Brick is on, cycle power.
6. After the boot sequence is completed, the following files will be generated under your PowerPmacIP folder:
   boot.log
   interfaces

The **interfaces** file includes all the network settings for the Power Brick, including the IP address.

If you want to change the Power PMAC network settings, follow these steps:

Modify the **interfaces** file created by Power PMAC under PowerPmacIP folder by connecting the USB memory stick/SD Card on PC, opening the file using any text editor which supports simple ASCII text, and modifying the settings you want to modify. Below is an example of the **interfaces** file:

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

iface eth0 inet static
address 10.34.9.232
netmask 255.255.255.0
gateway 10.34.9.254

iface eth1 inet static
address 192.168.0.232
netmask 255.255.255.0
gateway 192.168.0.232

auto eth0
auto eth1
```

1. Save the file and safely remove the USB memory stick/SD Card from the PC.
2. Plug the USB memory stick/SD Card into Power Brick's USB port.
3. If Power Brick is off, then power it up. If Power Brick is on, cycle power.
4. After the boot sequence is completed, Power Brick will have been updated with the new network settings.

Restoring Factory Default Configuration

Restoring Power Brick's settings to factory default can be done in two ways.

Method 1 (to be used when communicating):

Enter **$$$***** into the IDE Terminal Window. Issue a **SAVE**, followed by a **$$$** to maintain the factory default settings.

Method 2 (to be used when not communicating)

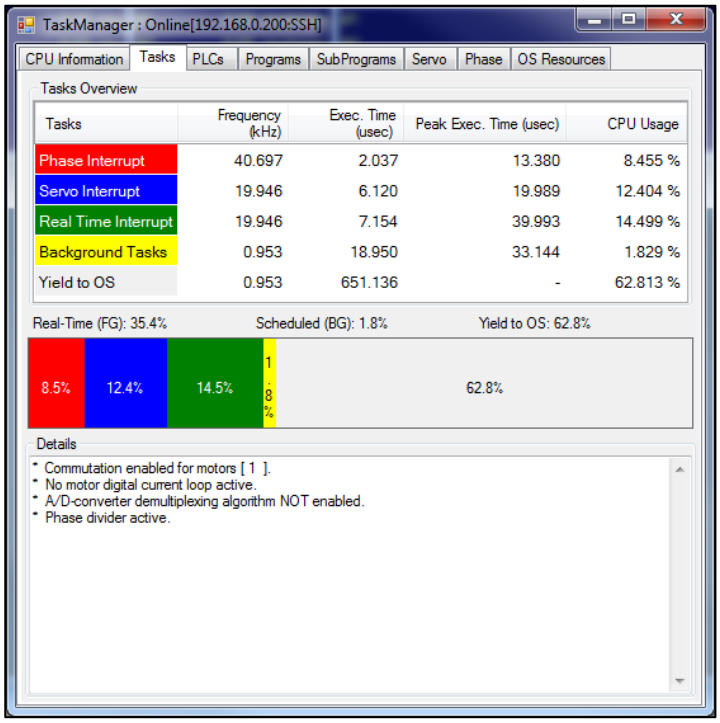
- Connect a USB memory stick or SD Card to a PC using any OS which can work with FAT32 partition.
- Create a folder named **PowerPmacFactoryReset** on the USB memory stick/SD Card root folder.
- Safely remove the USB memory stick/SD Card from the PC.
- Plug the USB memory stick/SD Card into Power PMAC's USB port.
- If Power Brick is off, then power up the device. If Power Brick is on, cycle power.
- After the boot sequence is completed, the Power Brick will be restored to factory default settings.

Watchdog Faults

Two types of Watchdog Faults can occur. The first is a "Soft Watchdog" which occurs when the CPU is starved of processing time and cannot reset the background (when **Sys.WDTFault = 2**) or the foreground (when **Sys.WDTFault = 1**) Watchdog timer. During normal operation, **Sys.WDTFault = 0**.

1. Avoid infinite loops in C programs because they can dominate the CPU and prevent background from resetting the Watchdog counter.

2. Check the CPU loading through the IDE by going to Tools→Task Manager, and then clicking the Tasks tab (see image below):



If the foreground ("Real-Time (FG)") loading is very high (e.g. 80%), you may encounter Watchdog faults at peak calculation times. You may need to reduce the Real-Time Interrupt time (i.e. set **Sys.RtIntPeriod** higher), reduce the servo/phase clock rates, or optimize your programming. If the Watchdog problem is intermittent, you may have a loading spike due to programming and you should revise your code. If the problem is not from code, it may be a hardware problem and the product will need to be sent in for repairs.

A "Hard Watchdog" occurs when the processor's Watchdog timer has been tripped, or when the processor does not receive proper logic power. All communication is lost in this case, the processor is shut down, all outputs are disabled, and the red "WD" LED on the front of the Power Brick activates. The only way to recover from this fault is to cycle power. If you receive this fault, check that +24 VDC is properly being applied to the logic power and has not sagged below +12 VDC. Since a "Hard Watchdog" can be caused by a CPU processing overload as well, the above troubleshooting steps also apply here.

BRICKLV STRUCTURE ELEMENTS

The **BrickLV** data structure elements consist of two main categories; global elements (**BrickLV.**) which affect all the channels and channel specific elements (**BrickLV.Chan[].**) which only affect the indexed channel. Each category (global or channel) consists of:

- Saved Setup Elements
- Non-saved Setup Elements (automatically reset)
- Status (read only)

The **BrickLV** data structure elements referred to in this section are "software" elements built into the Power PMAC firmware. They must not be confused with the ASIC (Gate 3) hardware elements **PowerBrick[]** and **PowerBrick[].Chan[]**.

Global Saved Setup Elements

BrickLV.MonitorPeriod

Description: Time interval for updating status registers

Range: 0 .. 4,294,967,295 (2^{32}-1)

Units: Milliseconds

Default: 0 (50 msec)

Legacy I-variable alias: none

BrickLV.MonitorPeriod tells Power PMAC software how much time there is between consecutive requests for the value of all Brick LV status registers. It is expressed in milliseconds as an integer value.

If **BrickLV.MonitorPeriod** is set to the default value of 0 or any value up to 50, all Brick LV status elements are updated every 50 milliseconds. Setting the value higher will reduce the update frequency and reduces the background time which monitor process takes from the Power PMAC CPU.

Note

The value of **BrickLV.MonitorPeriod** does not affect how often the amplifier stage checks the status conditions internally. It only controls how frequently the Power PMAC CPU requests this information.

While the value of **BrickLV.MonitorPeriod** is saved, the element that starts the monitoring process itself, **BrickLV.Monitor**, is not a saved setup element. It must explicitly be set to 1 by the user application in order to start the monitoring process. Also, when either the configuration process or the fault-clearing reset process is started with **BrickLV.Config** or **BrickLV.Reset**, respectively, the monitoring process is stopped, and it is not automatically restarted. The user application must explicitly restart the monitoring process.

The monitored data in the Power Brick LV is provided to the controller on the lower 10 bits of the **PowerBrick[*i*].Chan[*j*].AdcAmp[*k*]** registers and it is essential that **PowerBrick[*i*].Chan[*j*].PackInData** and **PowerBrick[*i*].Chan[*j*].PackOutData** are set to 0, disabling “packed” register access and allowing all ADC register bits to be read by the CPU.

Global Non-Saved Setup Elements

BrickLV.Config

Description: Amplifier configuration/initialization control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickLV.Config acts as a flag for the Power PMAC firmware which controls the initialization of Power Brick LV amplifier based upon the **BrickLV.** saved setup elements. The amplifier stage is not automatically configured at power-up, so the configuration process *must* be commanded explicitly by the user application before the amplifier stage can be used.

Setting **BrickLV.Config** to 1 in a Script command starts the initialization process as a background task on Power PMAC CPU. The element stays at the set value until either the initialization process is successfully completed, in which case the value of **BrickLV.Config** is set to 0, or until a configuration error is detected, in which case the **BrickLV.Config** value is set to a negative value indicating the error in the process. The following list shows the error codes which can be encountered:

Error Code	Description
-1	The assigned value is not accepted. Only a value of 1 or 0 can be assigned by user to this data structure.
-2	The **BrickLV.Monitor** was called while either the **BrickLV.Reset** or **BrickLV.Config** process was active.
-3	The configuration process was attempted on incompatible hardware. No amplifier hardware with the matching Power Brick part number was detected.
-4	No Power Brick hardware was detected. This error is generated if incompatible output stage is detected.
-7	The configuration process attempted used on incompatible hardware. No DPSGATE3 interface ASIC was detected.

If **BrickLV.Config** is set to 1 in an on-line command, there will be a text response indicating whether the configuration completed correctly or not, and if not, what the error was.

It is strongly recommended for users to confirm the pass/fail status of the initialization process whenever **BrickLV.Config** is set to a value of 1.


Note

While setting **BrickLV.Config** to 1 as part of the standard system initialization process after power-up will load the configuration parameters into the amplifier control circuitry, it is recommended instead to set **BrickLV.Reset** to 1, which will not only load the configuration parameters, but clear any faults that may have occurred due to power-on transient conditions.


Note

Setting **BrickLV.Config** to 1 to start the amplifier configuration process automatically stops the amplifier monitoring process, and the monitoring process does *not* automatically resume when the configuration is completed. **BrickLV.Monitor** must be set to 1 again in the user Script application to resume the monitoring process.

```
OPEN PLC ExamplePLC
Sys.WDTReset = 5000 / (Sys.ServoPeriod * 2.258) // Increase Foreground WD Timer Threshold
CALL DelayTimer.msec(250)                       // 250 msec delay

BrickLV.Config = 1
WHILE (BrickLV.Config > 0) {}
IF (BrickLV.Config != 0)
{
    // Take necessary action in case of a fault
    Sys.WDTReset = 0 // Restore Foreground WD timer Threshold
}
// Continue with script process
DISABLE PLC ExamplePLC
CLOSE
```

The process of waiting for the **BrickLV.Config** to execute in a PLC consumes a significant amount of background cycles and risks triggering a foreground soft watchdog fault (**Sys.WDTFault = 1**), especially with higher clock frequencies. Setting **Sys.WDTReset** temporarily to a larger value (increasing the foreground watchdog timer threshold) alleviates this issue.


Note

The **Sys.WDTReset** expression stated in the PLC example should ensure the proper setting regardless of the user specified clock frequencies.

BrickLV.Monitor

Description: Amplifier status monitoring update control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickLV.Monitor acts as a flag for the Power PMAC firmware which controls the execution of Power Brick LV amplifier status monitoring background task. This task updates the **BrickLV.** status elements at constant period set by saved setup element **BrickLVC.MonitorPeriod**.

If **BrickLV.Monitor** is set to its power-on default value of 0, there is no updating of the **BrickLV.** status elements. In this mode none of these element values are updated and they maintain their last updated value until next reset or power cycle.

Setting **BrickLV.Monitor** equal to 1 in a Script command starts the background **BrickLV.** status update task at a period set by **BrickLV.MonitorPeriod**. The element stays at the set value until either the user application sets the value to 0, which stops the update process, or the user application commands an initialization or reset process by setting **BrickLV.Config** or **BrickLV.Reset** to a value of 1.

If an error occurs during the monitor process, the **BrickLV.Monitor** value is set to a negative value indicating an error in the process. The following table shows the errors that can be reported. It is strongly recommended for users to confirm the pass/fail status of the monitoring initialization process whenever **BrickLV.Monitor** is set to a value of 1.

Error Code	Description
-1	The assigned value is not accepted. Only a value of 1 or 0 can be assigned by user to this data structure.
-2	The **BrickLV.Monitor** was was called while either the **BrickLV.Reset** or **BrickLV.Config** process was active.
-3	The configuration process was attempted on incompatible hardware. No amplifier hardware with the matching Power Brick part number was detected.
-4	No Power Brick hardware was detected. This error is generated if incompatible output stage is detected.
-6	Packed data mode is detected (**PowerBrick[*i*].Chan[*j*].PackInData** > 0). This error is only generated if the monitor process is requested.
-7	The configuration process attempted used on incompatible hardware. No DPSGATE3 interface ASIC was detected.


Note

The monitored data in the Power Brick LV amplifier is provided to the controller in the low bits of the **Gate3[*i*].Chan[*j*].AdcAmp[*k*]** registers, below the current feedback values. This data cannot be read if two phases are "packed" into one register, so it is essential that **Gate3[*i*].Chan[*j*].PackInData** and **Gate3[*i*].Chan[*j*].PackOutData** are set to 0, disabling packed data and allowing the full registers to be read by the CPU.


Note

The monitoring process is automatically halted when either **BrickLV.Config** or **BrickLV.Reset** is set to 1 to update the amplifier configuration or reset the amplifier state, respectively, with **BrickLV.Monitor** set to 0. The monitoring process is *not* automatically resumed when the configuration or reset process is finished, so it must be explicitly restarted when one of these other processes is finished.

```
OPEN PLC ExamplePLC
Sys.WDTReset = 5000 / (Sys.ServoPeriod * 2.258) // Increase Foreground WD Timer Threshold
CALL DelayTimer.msec(250)                              // 250 msec delay

BrickLV.Monitor = 1
CALL DelayTimer.msec(100)       // 100 msec delay
WHILE (BrickLV.Monitor > 0) {}
IF (BrickLV.Monitor < 0)
{
    // Take necessary action in case of a fault
    Sys.WDTReset = 0                // Restore Foreground WD timer Threshold
}
// Continue with script process
DISABLE PLC ExamplePLC
CLOSE
```

The process of waiting for the **BrickLV.Monitor** to execute in a PLC consumes a significant amount of background cycles and risks triggering a foreground soft watchdog fault (**Sys.WDTFault = 1**), especially with higher clock frequencies. Setting **Sys.WDTReset** temporarily to a larger value (increasing the foreground watchdog timer threshold) alleviates this issue.


Note

The **Sys.WDTReset** expression stated in the PLC example should ensure the proper setting regardless of the user-specified clock frequencies.

BrickLV.Reset

Description: Amplifier reset/fault-clear control

Range: -7 .. 1

Units: none

Power-on default: 0

BrickLV.Reset acts as a flag for the Power PMAC firmware which controls the reset process of Power Brick LV amplifier. This reset process clears any latched faults, and loads the configuration into the active amplifier-control circuits based upon the **BrickLV** saved setup elements.

Setting **BrickLV.Reset** equal to 1 in a Script command starts the reset process as a background task on Power PMAC CPU. The value stays at this set value until either the reset process is completed, in which case the value of **BrickLV.Reset** is set to 0, or an error occurs in which case the **BrickLV.Reset** value is set to a negative value indicating an error in the process. Please refer to **BrickLV.Config** for detailed information on the error code list.

It is strongly recommended for users to confirm the pass/fail status of the reset process whenever **BrickLV.Reset** is set to a value of 1.


Note

Setting **BrickLV.Reset** to 1 to start the amplifier configuration process automatically stops the amplifier monitoring process, and the monitoring process does *not* automatically resume when the configuration is completed. **BrickLV.Monitor** must be set to 1 again in the user Script application to resume the monitoring process.

```
OPEN PLC ExamplePLC
Sys.WDTReset = 5000 / (Sys.ServoPeriod * 2.258) // Increase Foreground WD Timer Threshold
CALL DelayTimer.msec(250)                       // 250 msec delay
BrickLV.Reset = 1
while (BrickLV.Reset > 0) {}
if (BrickLV.Reset < 0)
{
   // Take necessary action in case of a fault
       Sys.WDTReset = 0                // Restore Foreground WD timer Threshold
}
// Continue with script process
DISABLE PLC ExamplePLC
CLOSE
```

The process of waiting for the **BrickLV.Reset** to execute in a PLC consumes a significant amount of background cycles and risks triggering a foreground soft watchdog fault (**Sys.WDTFault = 1**), especially with higher clock frequencies. Setting **Sys.WDTReset** temporarily to a larger value (increasing the foreground watchdog timer threshold) alleviates this issue.


Note

The **Sys.WDTReset** expression stated in the PLC example should ensure the proper setting regardless of the user specified clock frequencies.

Global Status Elements

BrickLV.BusOverVoltage

Description: DC bus overvoltage fault flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.BusOverVoltage** status bit indicates whether the DC bus voltage supplied to Power Brick LV is above a maximum threshold or not. It is set to 0 if the measured DC bus voltage is 80 VDC or less. It is set to 1 if the measured DC bus voltage is greater than 80 VDC.

BrickLV.BusOverVoltage is a fault flag. If this fault is detected, the amplifier-fault lines for all channels are set to the "true" state, causing a software fault condition on all Power PMAC motors commanding these channels. It is a transparent status bit; as soon as the measured voltage no longer exceeds 80 VDC, the value of this bit is cleared to 0. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command. This status bit is only updated if **BrickLV.Monitor** is set to 1.

Note

The amplifier will shut down with a fault on all channels when it detects an overvoltage condition regardless of whether software status bits are updated for the processor (**BrickLV.Monitor** = 1) or not.

BrickLV.BusUnderVoltage

Description: DC bus under voltage warning flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.BusUnderVoltage** status bit indicates whether the DC bus voltage supplied to Power Brick LV is above a minimum threshold or not. It is set to 0 if the measured DC bus voltage is 12 V or more. It is set to 1 if the measured DC bus voltage is less than 12 V.

BrickLV.BusUnderVoltage is a warning flag; there is no fault condition generated if it is set to 1. It is a transparent status bit; as soon as the measured voltage reaches 12 V again, the value of this bit is cleared to 0. This status bit is only updated if **BrickLV.Monitor** is set to 1.

BrickLV.OverTemp

Description: Power board over temperature flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.OverTemp** status bit indicates whether the measured temperature of the power board is above a maximum threshold or not. It is set to 0 if the measured board temperature is 70 °C or less. It is set to 1 if the measured board temperature is over 70 °C.

BrickLV.OverTemp is a fault flag. If this fault is detected, the amplifier-fault lines for all channels are set to the "true" state, causing a software fault condition on all Power PMAC motors commanding these channels. It is a transparent status bit; as soon as the measured temperature no longer exceeds 70 °C, the value of this bit is cleared to 0. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command. This status bit is only updated if **BrickLV.Monitor** is set to 1.

Note

The amplifier will shut down with a fault on all channels when it detects an over temperature condition regardless of whether software status bits are updated for the processor (**BrickLV.Monitor** = 1) or not.

Channel Saved Setup Elements

BrickLV.Chan[*j*].I2tWarnOnly

Description: I^2T protection-level control

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: none

BrickLV.Chan[*j*].I2tWarnOnly determines the course of action the amplifier hardware takes upon detection of an excess integrated current (I^2T) condition on the channel. If **BrickLV.Chan[*j*].I2tWarnOnly** is set to the default value of 0, then upon detection of a I^2T excess condition, an amplifier fault is generated, the motor is killed, the corresponding status bit is set, and the corresponding error code is displayed on the amplifier (Error Code **n.L**).

If **BrickLV.Chan[*j*].I2tWarnOnly** is set to a value of 1, the I^2T excess condition will be reported as a warning in the status register, but it will not generate a fault on amplifier.

The **BrickLV.Chan[*j*].I2tWarnOnly** value is sent to the active amplifier-control circuit upon setting one of the non-saved setup elements **BrickLV.Reset** or **BrickLV.Config** equal to 1 in a Script command. It does not take effect until then.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The integrated current (I^2T) calculations accessed by this element are performed in the amplifier stage of the Power Brick LV. These calculations are separate from those done by the Power PMAC software.

BrickLV.Chan[*j*].TwoPhaseMode

Description: Channel motor phase count control

Range: 0 .. 1

Units: Boolean

Default: 0

Legacy I-variable alias: none

BrickLV.Chan[*j*].TwoPhaseMode selects the operational output mode of the amplifier channel. If set to its default value of 0, the amplifier is set to 3-phase operational mode, using the U, V, and W output lines. This operational mode is mainly used with Y-wound or delta-wound brushless servo motors (but 3-phase stepper motors do exist).

If the **BrickLV.Chan[*j*].TwoPhaseMode** is set to a value of 1, the amplifier channel is placed in 2-phase operational mode, using the U and W output lines to drive the first phase, and the V and X output lines to drive the second phase. This operational mode is mainly used with 2-phase stepper motors (but 2-phase brushless servo motors do exist).

Note

If the channel is put in 2-phase output mode with **BrickLV.Chan[*j*].TwoPhaseMode**, the Power PMAC motor commanding the channel should also be put in two-phase mode by setting bit 0 (value 1) of **Motor[*x*].PhaseMode** to 1.

Note

DC brush motors, voice-coil motors, and other similar "two-lead" motors that do not require electronic commutation can be driven between the U and W output lines with either setting of this element. However, it is recommended in this case to leave **BrickLV.Chan[*j*].TwoPhaseMode** at its default value of 0, so less processing of the commanded PWM signals is required.

The **BrickLV.Chan[*j*].TwoPhaseMode** value is sent to the active amplifier control circuits upon setting **BrickLV.Config** to 1 in a Script command. The user can check the operational mode of each channel by setting the **BrickLV.Monitor** equal to 1 in a Script command and reading the **BrickLV.Chan[*j*].ActivePhaseMode** value.

Channel Status Elements

BrickLV.Chan[*j*].ActivePhaseMode

Description: Channel active output phase mode configuration

Range: 0 .. 1

Units: Boolean

The **BrickLV.Chan[*j*].ActivePhaseMode** status bit indicates whether the channel is presently configured for 3-phase output or 2-phase output. It is set to 0 if the channel is configured for 3-phase output on the U, V, and W motor lines. It is set to 1 if the channel is configured for 2-phase output, with one phase on the U and W motor lines, and the other on the V and X motor lines.

The phase configuration is determined by the value of saved setup element **BrickLV.Chan[*j*].TwoPhaseMode**, but the value of this saved element is not copied into the active amplifier control circuits until the amplifier is successfully reset and/or configured by setting **BrickLV.Reset** or **BrickLV.Config** to 1 in a Script command. This status element can be used to confirm whether the configuration was completed successfully or not. It is only updated if **BrickLV.Monitor** is set to 1.

BrickLV.Chan[*j*].I2tFaultStatus

Description: Channel I2T fault/warning flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.Chan[*j*].I2tFaultStatus** bit indicates whether an excessive integrated current (I^2T) condition is present on the channel or not. It is set to 0 if the integrated current value is not excessive; it is set to 1 if it is excessive. This status flag is only updated if **BrickLV.Monitor** is set to 1.

An excessive I^2T condition will generate a fault if saved setup element **BrickLV.Chan[*j*].I2tWarnOnly** is set to its default value of 0. It will not generate a fault if **BrickLV.Chan[*j*].I2tWarnOnly** is set to 1.

BrickLV.Chan[*j*].I2tFaultStatus is a transparent status bit and it will be cleared to 0 as soon as the integrated current value falls below the threshold again. However, any motor software fault conditions it creates are latched, and the motors must explicitly be re-enabled by command.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an I2T excess condition if **BrickLV.Chan[j].I2tWarnOnly** is set to 0 regardless of whether software status bits are updated for the processor (**BrickAC.Monitor** = 1) or not.

Note

The integrated current (I^2T) calculations accessed by this element are performed in the amplifier stage of the Power Brick LV. These calculations are separate from those done by the Power PMAC software.

BrickLV.Chan[*j*].OverCurrent

Description: Channel over current fault flag

Range: 0 .. 1

Units: Boolean

The **BrickLV.Chan[*j*].OverCurrent** status bit indicates whether the hardware over-current detector for the channel has sensed an instantaneous overcurrent or short-circuit state for the channel or not. It is set to 0 if it has not detected this state. It is set to 1 if it has detected this state. This status flag is only updated if **BrickLV.Monitor** is set to a value greater than 0.

Over-current fault detection in Power Brick LV is performed in hardware. Once over-current fault is detected, the fault status is latched. This fault can be cleared by setting **BrickLV.Reset** equal to 1. Any motor software fault conditions it creates are also latched, and the motors must explicitly be re-enabled by command after this fault is cleared.

The channel index ***j*** (= 0 to 7) is one less than the corresponding hardware channel number (= 1 to 8).

Note

The channel will shut down with a fault when it detects an over-current condition regardless of whether software status bits are updated for the processor (**BrickLV.Monitor** = 1) or not.

BrickLVVers

Description: Amplifier firmware version

Range: 0.0 .. 15.0

Units: none

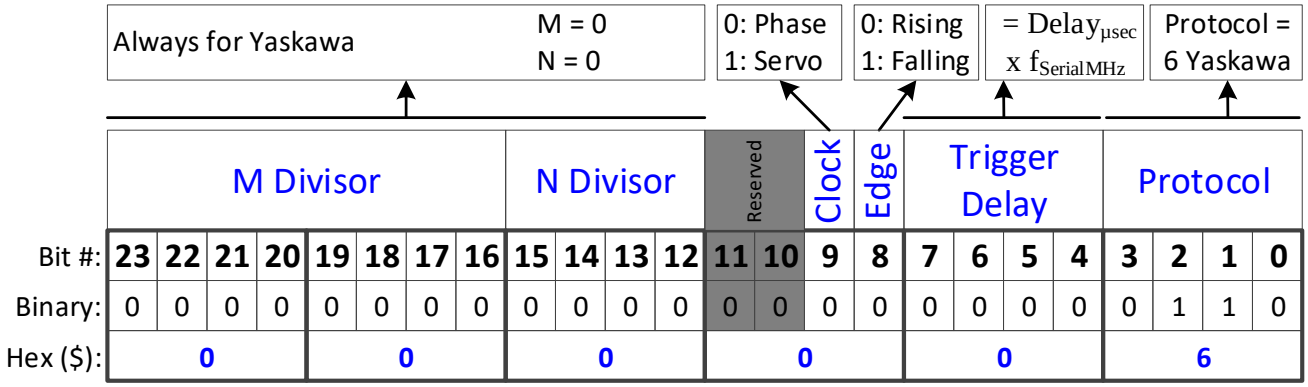
The **BrickLVVers** status element contains the amplifier firmware version (which is distinct from the Power PMAC CPU's firmware version) with a format of [**Version].[Release]** number. This element is only updated if **BrickLV.Monitor** is set to 1.

APPENDICES

Appendix A: Yaskawa ACC-84B Example

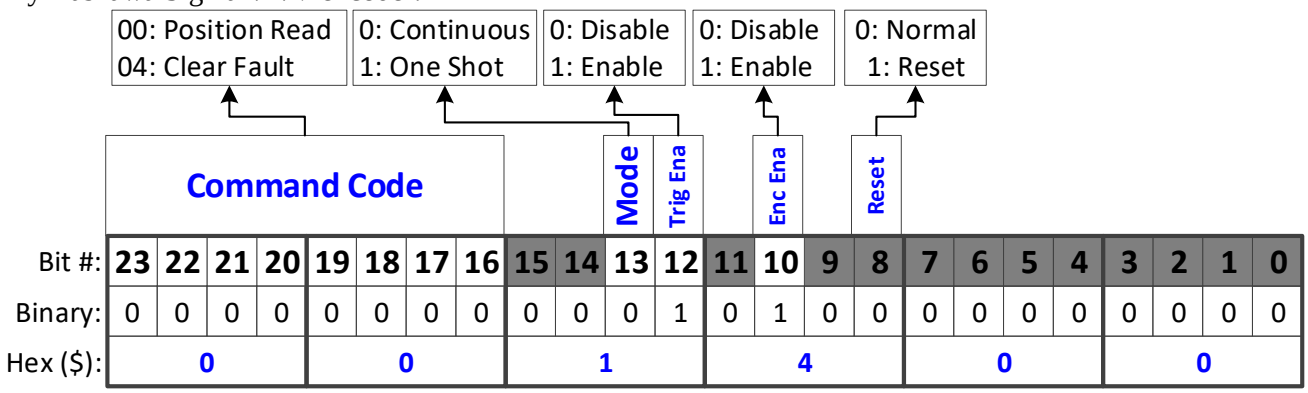
Serial Encoder Control Example– Yaskawa Sigma II/III/V

No trigger delay, rising edge of phase. For Yaskawa Sigma II/III/V protocols a serial frequency of 100 MHz is always used.



Serial Encoder Command Example – Yaskawa Sigma II/III/V

Any Yaskawa Sigma I/II/V encoder.

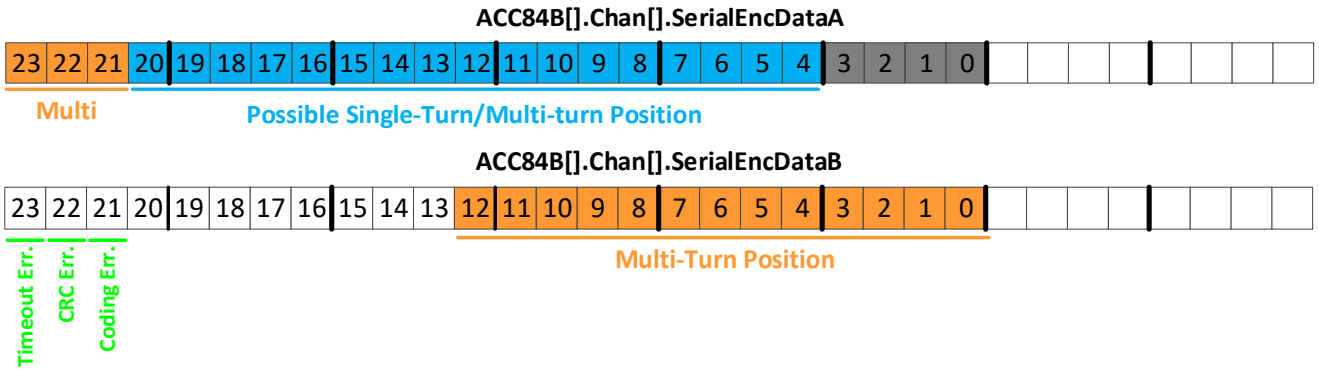



```
ACC84[0].SerialEncCtrl = $000006
ACC84[0].Chan[0].SerialEncCmd = $001400
```

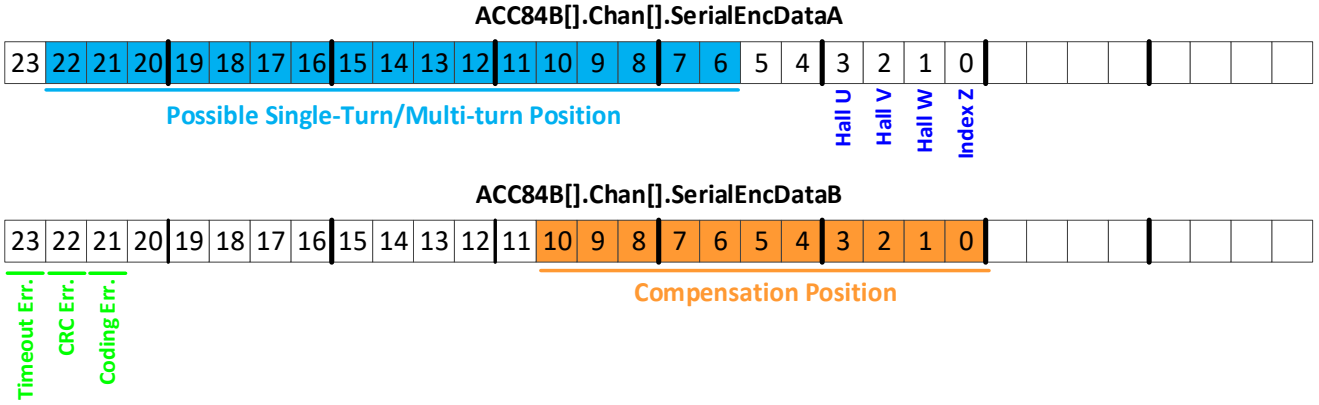
Serial Data Registers – Sigma II/III/V

For the Yaskawa Sigma II/III/V protocol, the data format in this element depends on the particular type of encoder.

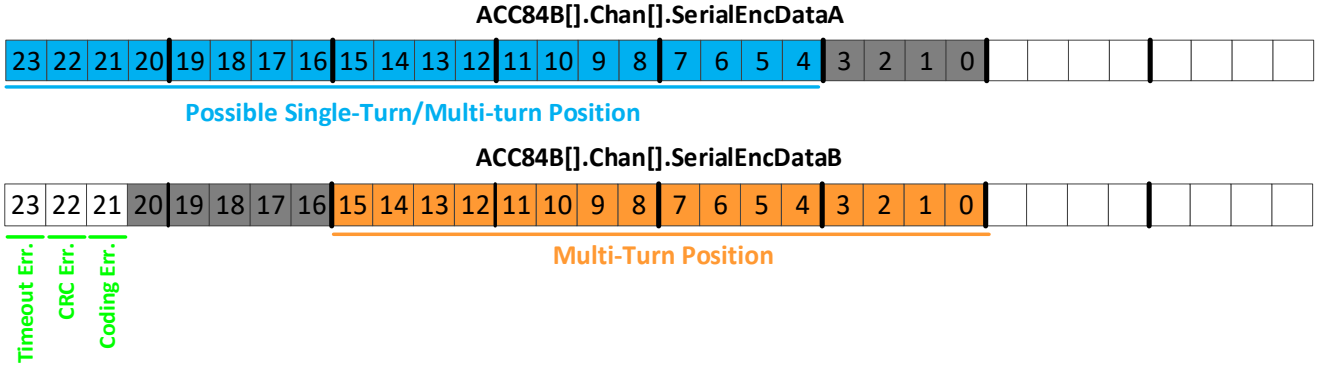
Yaskawa Sigma II (absolute 17-bit)



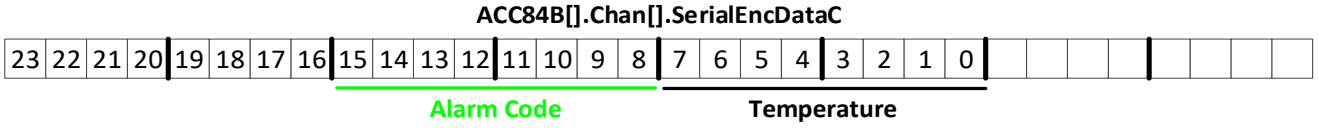
Yaskawa Sigma II (incremental 17-bit)



Yaskawa Sigma III/V (absolute 20-bit)



Additional Information



Yaskawa Sigma II/II/V Encoders Alarm Code (Absolute Encoders)

Script Bit #	C Bit #	Alarm Code
8	16	Backup Battery Alarm
9	17	Power-on error self-detected
10	18	Battery Level Warning
11	19	Absolute Error
12	20	Over Speed
13	21	Overheat
14	22	Position reference (index) not found yet

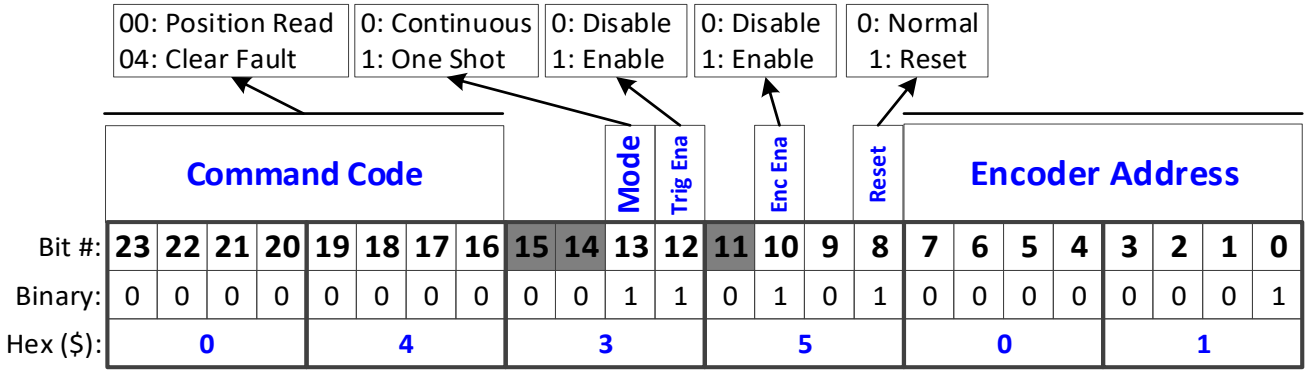
Yaskawa Sigma II/II/V Encoders Alarm Code (Incremental Encoders)

Script Bit #	C Bit #	Alarm Code
9	17	Power-on error self-detected
11	19	Revolution count (index to index) incorrect
14	22	Position reference (index) not found yet

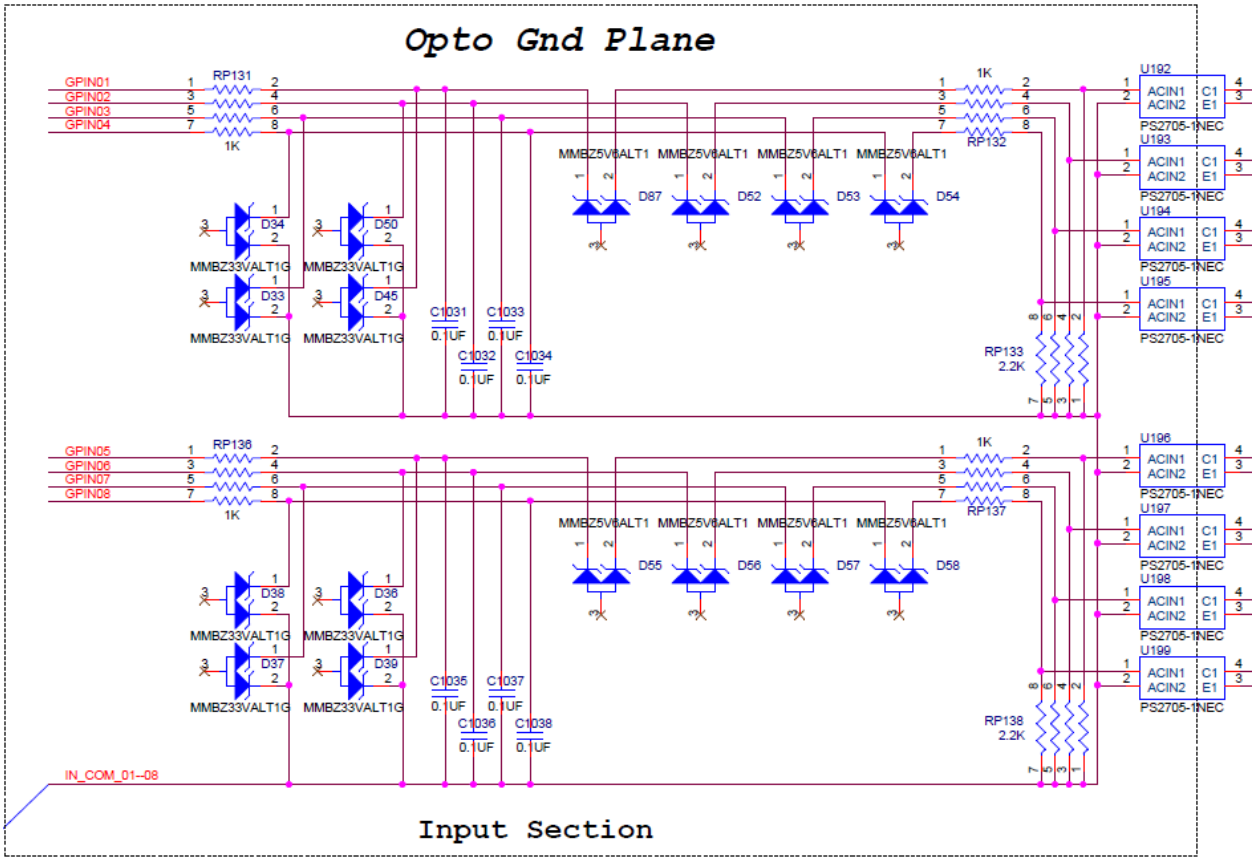
Resetting Faults – Yaskawa Sigma II/III/V

The following steps show the procedure for clearing the latched alarms on absolute encoders which the user/plc should perform in certain order:

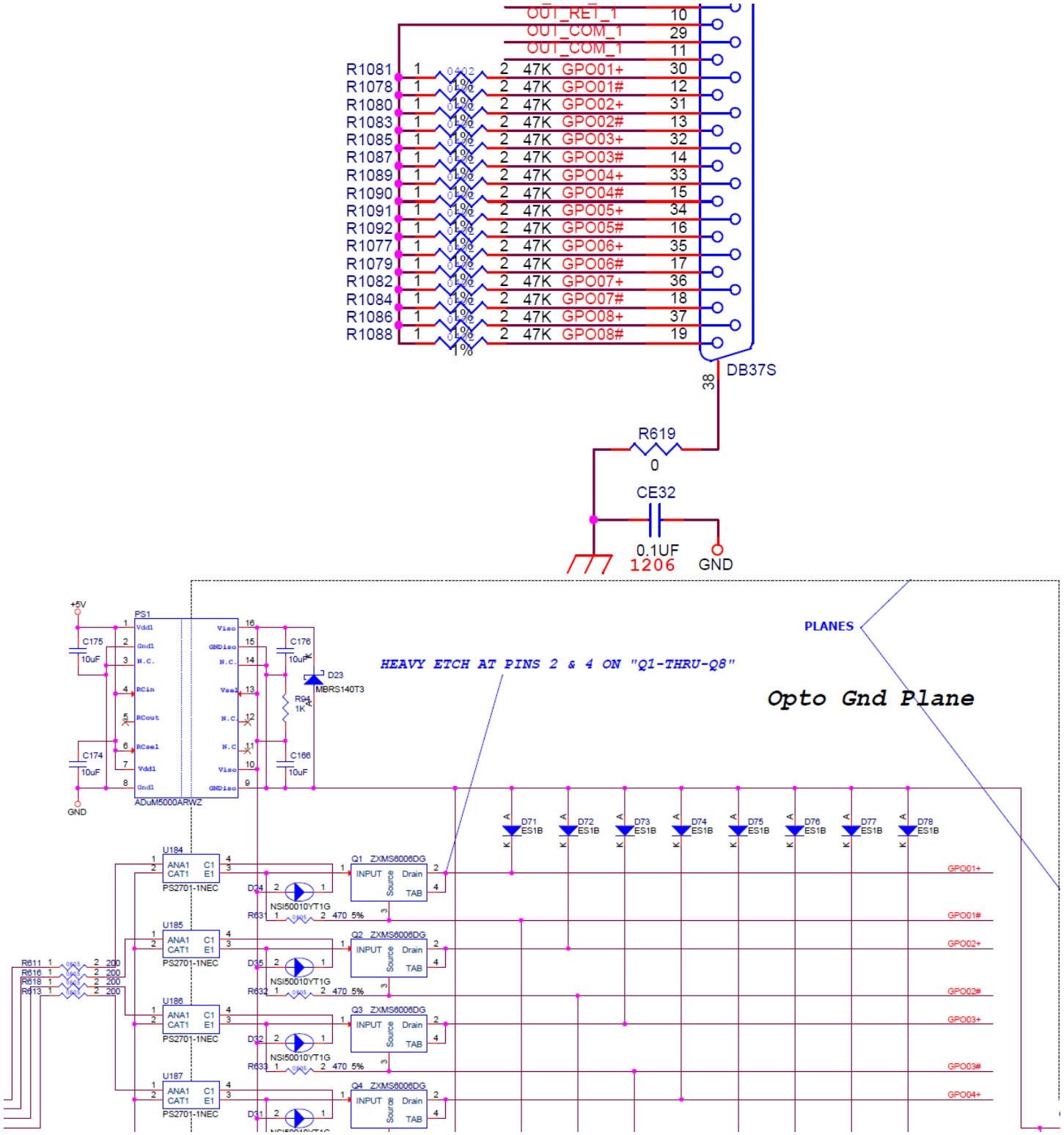
1. Write the value $043501 to **Acc84B[i].Chan[j].SerialEncCmd**.
2. Wait 10 milliseconds.
3. Wait for the trigger-enable component (Script bit 12) of this element to clear.
4. Wait for the busy signal (Script bit 8) of **Acc84B[i].Chan[j].SerialEncDataB** to clear. If cleared go to step 7.
5. Clear the command code of this element to $00 by writing $003501 to the element.
6. Repeat steps 2 to 4.
7. Resume continuous position requests by writing $001400 to the element.



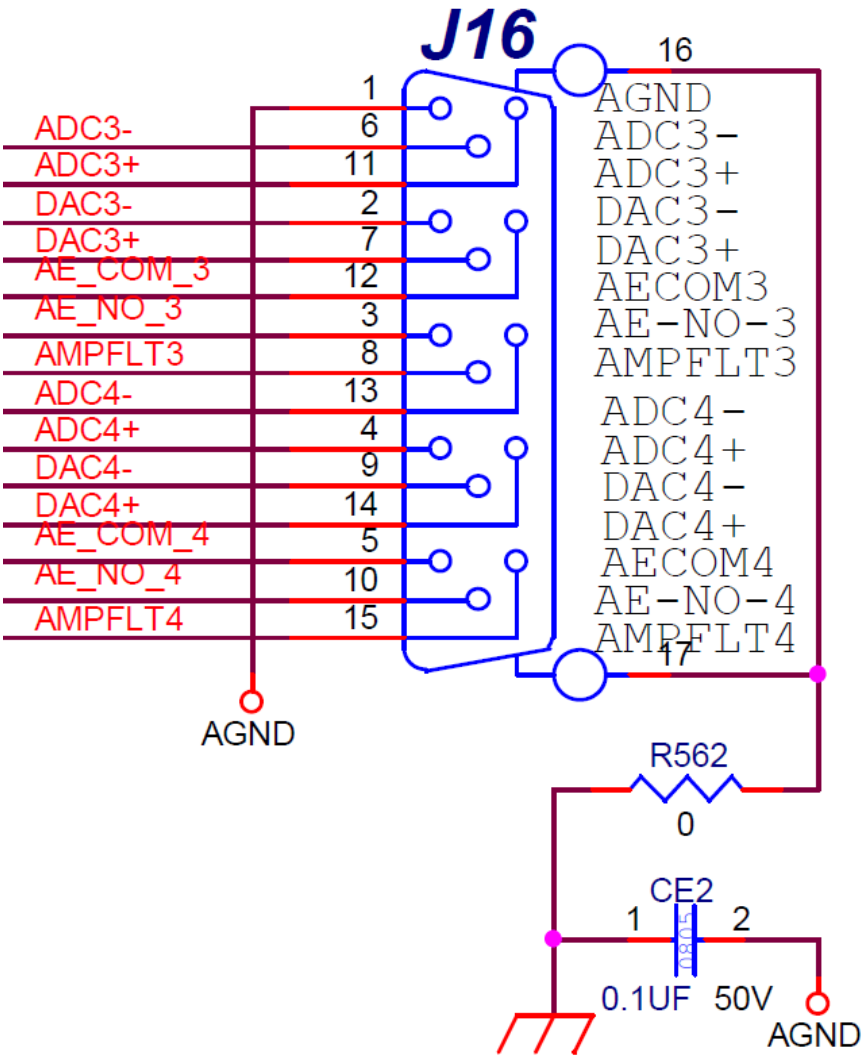
Appendix B: Digital Inputs Schematic

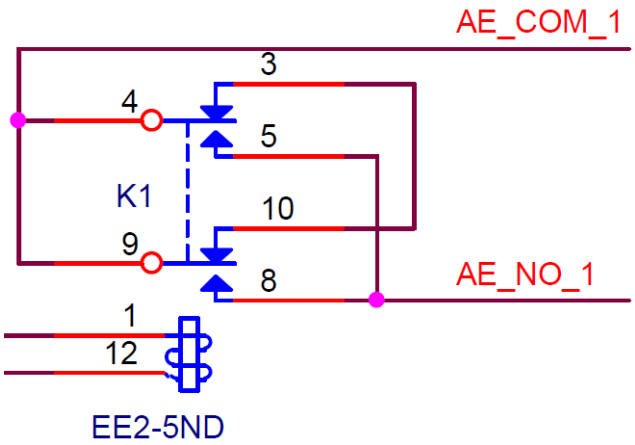


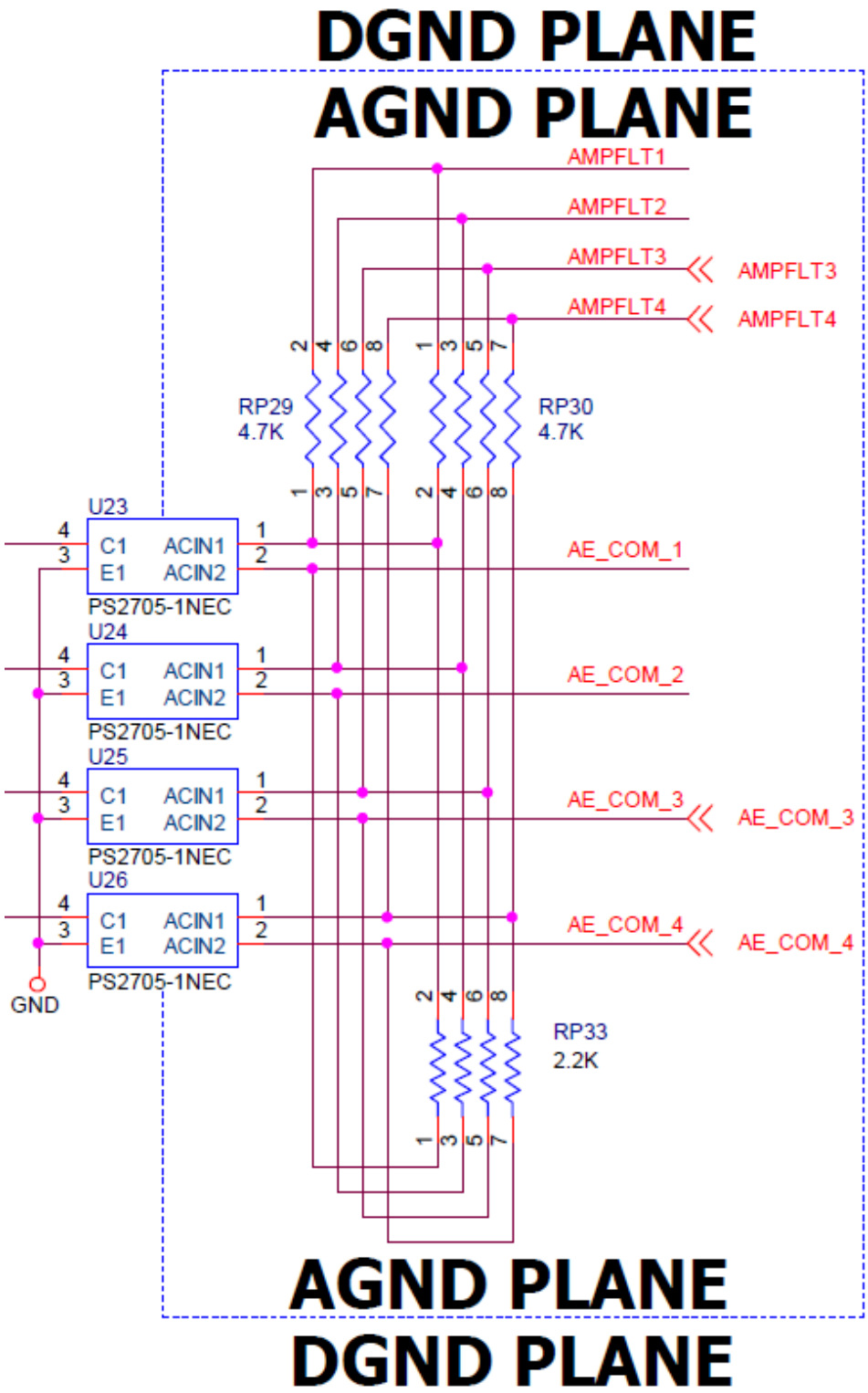
Appendix C: Digital Outputs Schematic



Appendix D: Analog I/O Schematics






DGND PLANE
AGND PLANE
AMPFLT1
AMPFLT2
AMPFLT3
AMPFLT3
AMPFLT4
AMPFLT4
RP29
4.7K
RP30
4.7K
U23
C1
E1
ACIN1
ACIN2
PS2705-1NEC
U24
PS2705-1NEC
U25
PS2705-1NEC
U26
PS2705-1NEC
AE_COM_1
AE_COM_2
AE_COM_3
AE_COM_3
AE_COM_4
AE_COM_4
GND
RP33
2.2K
AGND PLANE
DGND PLANE

Appendix E: Limits & Flags Schematic