Making Your Power PMAC Application Safe

Important note: Delta Tau Data Systems has provided many safety features on the Power PMAC controller, and invested many resources to make Power PMAC a safe product. However, the ultimate responsibility for the safety of a control system using Power PMAC must lie with the system designer, utilizing the safety features on Power PMAC and in other parts of the system.

Watchdog Timer

Power PMAC has an on-board "watchdog timer". This subsystem provides a fail-safe shutdown to guard against many types of software and hardware malfunction. To keep it from tripping, the hardware circuit for the watchdog timer requires that two base conditions be met. First, the nominal 5-volt DC power supply must be greater than approximately 4.75V. If the supply voltage is below this value, the circuit will trip and the Power PMAC system will go into a "hard" watchdog failure. This feature is intended to prevent corruption of registers due to insufficient supply voltage.

The second necessary condition is that the timer must see a square-wave input (provided by the Power PMAC software) of a frequency greater than approximately 20 Hz, which means the digital signal must be toggled more than 40 times per second. In the foreground, the real-time interrupt routine decrements a counter (as long as the counter value is greater than zero), causing the least significant bit of the timer to toggle. This bit is fed to the timer itself.

At the end of each background cycle, after one scan of one active background Script PLC program and one scan of all active background C PLC programs, the background software resets the counter to the value specified by saved setup element **Sys.WDTReset**. (If this is set to a value less than 100, the counter is reset to 5000 each background cycle.)

Soft Watchdog Trips

If the processor is able to detect certain failures of the routines that support the watchdog timer, it will execute a "soft" watchdog trip. In a soft watchdog trip, all programmed motions are aborted, all motors are killed, and all interface hardware is locked into its reset state, which forces discrete outputs to their "off" state and continuous outputs (e.g. DAC, PWM, PFM) to their zero state. However, the processor continues to operate, and it can still communicate with a host computer. The hardware of the watchdog timer circuit is disabled so it cannot shut down the processor completely. The electrical power to the system must be removed and re-applied in order to clear a soft watchdog trip.

The purpose of a soft watchdog trip is to detect certain conditions that would likely lead to a hard watchdog trip and provide a safe shutdown of the system while keeping the processor alive so it is easier to figure out what the underlying problem is and to fix it. Soft watchdog trips are usually caused by user configurations that do not permit all tasks to execute in a timely fashion.

The first condition that causes the execution of a soft watchdog trip occurs if the processor sees that the counter decremented by the real-time interrupt routine has reached a value of zero. In this case, it will set global status element **Sys.WDTFault** to 1. A failure of this type typically means that there is inadequate processor time remaining for background tasks, so the time between background cycles is too large. The threshold time for this type of failure can be adjusted by changing the setting of **Sys.WDTReset**.

The second condition that causes the execution of a soft watchdog trip occurs if several consecutive background cycles execute without a real-time interrupt occurring in between to decrement the counter value. In this case, it will set global status element **Sys.WDTFault** to 2. A failure of this type typically means that the interrupts have failed or been set with too large a period (too low a frequency). Saved setup element **Sys.BgWDTReset** specifies the number of background cycles that can execute without an intervening real-time interrupt before a trip. Note that if it is set too large, it may not be able to detect this type of condition before a hard trip occurs.

Note that if no phase or servo clock interrupt signals are detected by the processor during power-up/reset, Power PMAC goes into a special "stay alive" mode executing background software only. This mode is similar to, but distinct from, a soft watchdog trip. No motors may be enabled in this mode, but the interface hardware is still active, permitting the user to configure the clock signal sources correctly. However, if the clock signal interrupts were present at power-up/reset and subsequently lost, a soft watchdog trip would occur because no real-time interrupt algorithms would execute to decrement the counter.

Hard Watchdog Trips

If the hardware watchdog timer circuit detects either an undervoltage or an underfrequency condition (which means that the software algorithms could not anticipate the condition and cause a soft trip), a "hard" watchdog trip will occur. In a hard watchdog trip, all interface hardware is locked in its reset state, which forces discrete outputs to their "off" state and continuous outputs (e.g. DAC, PWM, PFM) to their zero state. In addition, the processor itself is completely shut down, so no communications is possible.

In the event of a hard watchdog trip, the solid-state watchdog relay on the Power PMAC CPU board toggles. The "normally open" contact opens (as it does when no power is present) and the "normally closed" contact closes (again, as it does when no power is present). The system designer can make use of this relay output as part of a "safety string" to make sure output devices are properly disabled on a watchdog trip.

A hard watchdog trip is usually caused by a fundamental hardware problem that permits neither foreground (interrupt) nor background tasks to operate properly, so a soft trip is not possible. The electrical power to the system must be removed and re-applied in order to (try to) clear a soft watchdog trip.

Following Error Limits

"Following error" is simply the difference between a motor's net commanded position and its net actual position at any given time. All applications will have non-zero following error on their motors some (or most) of the time. The purpose of the servo loop is to try to drive this error to zero, but it will not succeed perfectly.

While some following error is always to be expected in an application, sufficiently large following errors can be indicative of serious, and often very dangerous problems, such as loss of feedback, reversed feedback, or mechanical failure.

Fatal Following Error Limit

If the magnitude of a motor's following error exceeds the limit set by **Motor[*x*].FatalFeLimit**, Power PMAC will automatically "kill" that motor. When a motor is "killed", its servo loop is

opened, the servo output is forced to zero, and the amplifier is disabled. The status bit **Motor[*x*].FeFatal** is set to 1 on this event; this bit is not cleared until the motor is disabled.

On detecting an fatal following-error condition, if bit 0 of **Motor[*x*].FaultMode** is set to the default value of 0, other motors in the coordinate system (even if they just have the "null" definition – **#*x*->0** – in that coordinate system) are automatically "aborted" (controlled deceleration to enabled, closed-loop stop). If bit 0 of **Motor[*x*].FaultMode** is set to 1, other motors in the coordinate system are automatically "killed" as well.

The coordinate system status bit **Coord[*x*].FeFatal** is set to 1 if the comparable motor status bit is set to 1 for any motor in the coordinate system. Motors in other coordinate systems are not affected.

If the coordinate system was executing a motion program at the time, the program execution would be aborted as well. Aborting a motion program stops program calculations, resets the program counter to the beginning, and discards any already computed motion equations from the queue. Execution cannot simply be resumed at the aborted point (this point in the program can be determined by use of the **list apc** on-line query command).

Motor[*x*].FatalFeLimit is expressed in the user's motor units. The magnitude of these units is determined by the feedback resolution, the encoder table entry's scaling factor **EncTable[*i*].ScaleFactor**, and the motor's position-loop scaling factor **Motor[*x*].PosSf**. Most users will scale the motor to units of "counts" or "LSBs" of the feedback sensor. However, others may wish to use (much larger) engineering units such as millimeters, inches, or degrees. If the user changes the definition of the motor units, the physical size of the fatal following error limit is automatically changed. When changing from small to large motor units, the limit may be increased so much as to be ineffective.

This limit may be disabled by setting **Motor[*x*].FatalFeLimit** to zero, or effectively disabled by setting it very large, but this is strongly discouraged in any application that has the potential to kill or injure people, or even to cause property damage. Disabling the fatal limit removes an important protection against serious fault conditions that can cause runaway situations, bringing the system to full power output faster than anybody could react.

Good tuning of your motor's servo loop is important for safety reasons as well as performance reasons. The smaller you can make your true following errors during proper operation, the tighter you can set your fatal following error limits without getting nuisance trips. Particularly important in this regard are the feedforward terms that can dramatically reduce the errors at high speeds and accelerations.

Warning Following Error Limit

If the magnitude of a motor's following error exceeds the limit set by **Motor[*x*].WarnFeLimit**, Power PMAC will automatically set the motor status bit **Motor[*x*].WarnFe** and the coordinate-system status bit **Coord[*x*].WarnFe**. These are "transparent" status bits; as soon as the magnitude of the motor's following error falls below the limit, the motor status bit is cleared. (The coordinate system status bit is simply the logical "or" of all of the motor status bits in the coordinate system.)

If **Motor[*x*].CaptureMode** is set to 2, the status bit **Motor[*x*].WarnFe** will be used as the trigger flag for Power PMAC's automatic triggered moves (homing-search moves, jog-until-trigger,

programmed rapid-mode move-until-trigger) instead of the default input trigger. This permits easy implementation of tasks such as homing into a hard stop, torque-limited screwdriving, etc.

Position (Overtravel) Limits

Power PMAC has both software and hardware "overtravel" position limit features. These are intended to prevent motion accidentally commanded out of the legal range of positions.

Software Overtravel Limit Parameters

Power PMAC also has positive and negative software limit parameters for each motor. These can be used to complement or replace the hardware limits. Saved setup elements **Motor[*x*].MaxPos** and **Motor[*x*].MinPos** define the positive and negative software position limits, respectively, in motor units, for each motor. These limits are referenced to the motor's zero (home) position, and do not change if the programming origin for the associated axis is offset. The value of **Motor[*x*].MaxPos** must be greater than that of **Motor[*x*].MinPos** for these software limits to be active. By default, both are set to 0.0, disabling them.

Action on Closed-Loop Trip

Power PMAC continually compares the motor's actual position to these limits. If the actual position of the motor exceeds the legal range set by these limits during a closed-loop move, Power PMAC automatically "aborts" the motor. Aborting a motor causes a controlled deceleration to a closed-loop zero-velocity state, using the saved setup elements **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**. If these values are positive, they represent the overall deceleration time and the S-curve time, respectively, in milliseconds. If these values are negative they represent the inverse of the deceleration rate (in $msec^2$ / motor unit) and inverse of the S-curve jerk rate (in $msec^3$ / motor unit).

Action on Open-Loop Trip

If the actual position of the motor exceeds the legal range set by these limits during an open-loop move, the action is dependent on the setting of saved setup element **Motor[*x*].FaultMode**. If bit 1 (value 2) is set to the default value of 0, Power PMAC "aborts" the motor. This closes the servo loop with the initial commanded velocity being equal to the present actual velocity, and causes a controlled deceleration to a stop, just as if the loop were closed when the limit switch was encountered. However, if bit 1 is set to 1 when the motor passes a software limit in open-loop mode, Power PMAC "kills" (disables) the motor. This immediately causes a zero output command and disables the amplifier.

Anticipating a Software Limit Trip

When possible, Power PMAC also compares the motor's desired destination position to these limit values as well. If the software overtravel limits are active, Power PMAC automatically converts an indefinite positive jog command (**`j+`**, **`jog+`**) to a definite jog to the positive limit, and an indefinite negative jog command (**`j-`**, **`jog-`**) to a definite jog to the negative limit. This means that the jog move will stop at the limit value, not begin to stop as it passes the limit value.

The software position limits are automatically disabled during homing search moves until the homing trigger is found. As soon as the trigger is found, the software limits are re-activated, using the new home position as the reference.

If the coordinate system setup element **Coord[*x*].SoftLimitStopDis** is set to the default value of 0, when a motor in the coordinate system hits a software limit during a programmed move, the

motion program is aborted, and all of the motors in the coordinate system are aborted as well. However, if **Coord[*x*].SoftLimitStopDis** is set to 1, motion program execution will continue, and the other motors in the coordinate system will continue to move while the offending motor is stopped at the limit position as long as the commanded position is beyond the limit position. In this mode, the software limits act as position saturation points, not error limits.

With the special lookahead buffer active (lookahead buffer defined, **Coord[*x*].LHDistance** > 0), then for segmented moves in the coordinate system (linear, circle, PVT mode moves with **Coord[*x*].SegMoveTime** > 0), Power PMAC checks the destination position of each motor for each segment computed against the software limits at lookahead time. If it finds a violation, it will work backwards through the lookahead buffer to compute a deceleration to a stop at the limit for the offending motor that is within the acceleration limits for the motor. Depending on the setting of **Coord[*x*].SoftLimitStopDis**, it may also stop the other motors simultaneously.

Hardware Overtravel Limit Switches

The axis-interface circuitry associated with each servo interface channel in a Power PMAC system has positive and negative hardware overtravel limit switch inputs. The exact nature of this input circuitry and instructions for connecting the limit switches are described in the Hardware Reference manual for each Power PMAC and axis-interface accessory.

Limit Switch Interface Circuitry

Generally, these inputs are optically isolated, with a failsafe circuit design. The limit switches must be "normally closed", conducting current through the opto-isolator when the axis is not in the limit. (Usually these are "AC optos" that can conduct current in either direction, permitting sinking or sourcing limit switches.) This conducting condition produces a "zero" state in the flag register for the channel in the Servo IC; the processor must read this zero to permit motion in that direction. To ensure failsafe operation, this polarity cannot be changed by the user.

Anything that stops current from flowing through the opto-isolator, whether from actually hitting the limit, from cable disconnection, or from loss of power supply for the limit circuit, produces a "one" state in the Servo IC. When the processor sees this, it will not permit motion in that direction.

Saved setup element **Motor[*x*].pLimits** for the motor must contain the address of the flag register for the channel into which these limit switches are wired (usually **Gate*n*[*i*].Chan[*j*].Status.a**). If **Motor[*x*].pLimits** is set to 0, the hardware overtravel limit function for the motor is disabled. Some users will want to do this permanently, as for a continuously rotating rotary axis; others will want to do this temporarily, as when homing into a limit switch.

Saved setup element **Motor[*x*].LimitBits** specifies which 2 bits of the 32-bit register specified are read for the status of the limit inputs. This should be set to 25 when using a PMAC2-style "DSPGATE2" IC, as on an ACC-24E2x UMAC board, or to 25 when using the standard MACRO-ring protocol. It should be set to 9 when using a PMAC3-style "DSPGATE3" IC, as on an ACC-24E3 UMAC board.

Action on Closed-Loop Trip

The limit input signals are direction sensitive for closed-loop moves: the positive-end limit pin only stops positive direction moves (those coming at it from the negative side), and the negative-end limit pin only stops negative direction moves (those coming at it from the positive side). This

makes it possible to command a move out of the limit that you have run into. However, this also makes it essential to have your limit switches wired into the proper inputs, or they will be useless.

Controlled Stop

When a motor hits a limit when the servo loop is closed, the action is dependent on the setting of bit 2 (value 4) of **Motor[*x*].FaultMode**. If this bit is set to its default value of 0, Power PMAC automatically "aborts" the motor. Aborting a motor causes a controlled deceleration to a closed-loop zero-velocity state, using the saved setup elements **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**. If these values are positive, they represent the overall deceleration time and the S-curve time, respectively, in milliseconds. If these values are negative they represent the inverse of the deceleration rate (in $msec^2$ / motor unit) and inverse of the S-curve jerk rate (in $msec^3$ / motor unit).

If the motor hit the limit during a motor move such as a jog move, the other motors in the coordinate system are not affected. However, if the motor hit the limit during a motion program commanded move, the motion program is stopped and all of the motors in the coordinate are aborted as well. Note that if the coordinate system has been executing a path move, this deceleration will not necessarily be along that path. Motors in other coordinate systems are not affected in either case.

Disabled Stop

However, if bit 2 (value 4) of **Motor[*x*].FaultMode** is set to 1, then this motor is "killed" (open-loop, zero command output, amplifier disabled) on hitting a hardware limit. (Note, however, that if the motor is already in an "abort" deceleration from exceeding a software overtravel limit, no further action will be taken when the hardware limit is hit.) Other motors are affected just as for an "abort" stop of this motor.

The behavior of this mode of operation is based on the idea that the software limits should be used to catch controlled excursions out of the intended range of operation, as when too large a position is commanded. In this case, a controlled stop is quicker, covers a shorter distance, and is easier to recover from. The hardware limit switches, which should be set just outside the software limit positions, are used to catch uncontrolled excursions out of the intended range of operation, when there is a problem with feedback so the software limits do not work.

Action on Open-Loop Trip

The limit input signals are not direction sensitive for open-loop moves: hitting either limit switch on any open-loop move of either sign (or zero) will cause a limit fault.

Controlled Stop

The user has a choice for what happens when a motor hits a limit switch when the servo loop is open. If bit 1 (value 2) of saved setup element **Motor[*x*].FaultMode** is set to the default value of 0, Power PMAC "aborts" the motor. This closes the servo loop with the initial commanded velocity being equal to the present actual velocity, and causes a controlled deceleration to a stop, just as if the loop were closed when the limit switch was encountered.

Disabled Stop

However, if bit 1 (value 2) of **Motor[*x*].FaultMode** is set to 1 when the motor hits a limit switch in open-loop mode, Power PMAC "kills" (disables) the motor. This immediately causes a zero output command and disables the amplifier.

In either case, other motors, whether in the same coordinate system or in a different coordinate system, are not affected.

Encoder Loss Detection

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 system is – usually it thinks it is "stuck" – and it can react wildly, often causing a runaway condition. Many of the safety checks performed, such as fatal following error limits, integrated current limits, overtravel limit switches, and amplifier fault signals, may be ineffective in this type of situation, reacting too late, or not at all. For this reason, many people want the capability to directly monitor the presence of the feedback signal.

Many of Power PMAC's servo interfaces have circuitry dedicated to monitoring the presence of a proper feedback signal. In addition, Power PMAC can automatically check these circuits for loss of sensor signal and take appropriate shutdown action.

Signal Loss Detection Circuits

Different types of feedback sensors require different circuits to monitor for loss of the signal, or at least loss of a valid signal. Power PMAC interfaces provide circuits for several of the most common types of feedback sensors.

Digital Quadrature Encoders

Digital quadrature encoders are one of the most common types of feedback sensors used in motion control systems. Almost always, the individual channel signals are differential pairs at 5V levels. Most Power PMAC interfaces for these encoders have circuitry that checks for the presence of a proper differential signal pair on each signal channel, utilizing "exclusive-OR" (XOR) logic gates, which output a high level for a proper differential signal, when the two inputs to the gate are at different logical levels.

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. These gate outputs from the A and B encoder channels are logically combined to create a single present/lost signal for the encoder (if either channel is invalid, it will indicate "lost"). For interfaces using the PMAC2-style DSPGATE1 ASIC, such as the ACC-24E2*x* UMAC axis-interface boards, this flag is found in the low-true element **Gate1[*i*].Chan[*j*].EncLossN**. For interfaces using the PMAC3-style DSPGATE3 ASIC, such as the ACC-24E3 UMAC axis-interface boards and the Power Brick products, this flag is found in the high-true element **Gate1[*i*].Chan[*j*].LossStatus**.

Analog Sinusoidal Encoders and Resolvers

Analog sinusoidal encoders and resolvers provide simultaneous "sine" and "cosine" signals into the analog-to-digital converters of the Power PMAC interface circuitry for these devices. 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 PMAC3-style DSPGATE3 ASIC used on the ACC-24E3 UMAC axis-interface board and in the Power Brick products automatically computes this sum-of-squares value every sample cycle. The latest value is always available in the 16-bit element **Gate3[*i*].Chan[*j*].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 **Gate3[*i*].Chan[*j*].SosError** is automatically set to 1.

Serial Encoders

Power PMAC 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. Note that the SSI and SPI protocols cannot provide this detection.

If the serial-encoder interface of the PMAC3-style DSPGATE3 ASIC used on the ACC-24E3 UMAC axis-interface board and in the Power Brick products is used, this "timeout error" flag is bit 31 of the element **Gate3[*i*].Chan[*j*].SerialEncDataB**. If the FPGA-based ACC-84E UMAC serial-encoder-interface board is used, this flag is bit 31 of the element **Acc84E[*i*].Chan[*j*].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 PMAC interfaces for serial encoders can evaluate these mechanisms and determine whether the data set was valid or not. This is particularly recommended for the SSI and SPI protocols, where the data patterns cannot be used to detect a timeout error.

For the SSI protocol, the parity error flag is bit 31 of **Gate3[*i*].Chan[*j*].SerialEncDataB** or **Acc84E[*i*].Chan[*j*].SerialEncDataB**. For the SPI protocol, any error reporting is vendor specific, but would be found (if it exists) in a high bit of one of these registers.

Software Setup for Loss Detection

Power PMAC permits automatic checking for sensor loss on each motor, and if loss is detected, an immediate shutdown action. There are four saved setup elements for each motor to configure this functionality:

- **Motor[*x*].pEncLoss** Address of register with sensor loss flag
- **Motor[*x*].EncLossBit** Bit number of sensor-loss bit in **pEncLoss** register
- **Motor[*x*].EncLossLevel** Sensor-loss logical state
- **Motor[*x*].EncLossLimit** Sensor-loss maximum number of fault detections

Note that if **Motor[*x*].pEncLoss** is set to its default value of 0, loss detection is disabled for the motor. Therefore, there is no automatic sensor-loss detection by default. The user must explicitly configure it in an application.

Typical settings of these elements for the common types of feedbacks are discussed below.

Digital Quadrature Encoders

For digital quadrature encoders connected to an ACC-24E2*x* UMAC axis-interface board with a PMAC2-style DSPGATE1 ASIC, the following settings should be used.

```
Motor[x].pEncLoss = Gate1[i].Chan[j].EncLossN.a  // Encoder-loss register
Motor[x].EncLossBit = 13                         // Loss bit number
Motor[x].EncLossLevel = 0                        // Low-true fault
```

Note that instead of the **Gate1[*i*]** structure name, it is also possible to use the alias name for the particular board: **Acc24E2[*i*]**, **Acc24E2A[*i*]**, or **ACC24E2S[*i*]**. Socketed resistor packs on the

circuit boards must be reversed from their default orientation to enable the creation of the loss signal in the accessory. Consult the Hardware Reference Manual for the accessory for details.

For digital quadrature encoders connected to an ACC-24E3 UMAC axis-interface board with a PMAC3-style DSPGATE3 ASIC, or to a Power Brick control board, the following settings should be used:

Motor[*x*].pEncLoss = **Gate3[*i*].Chan[*j*].LossStatus.a** // Encoder-loss register
Motor[*x*].EncLossBit = 28 // "Transparent" loss bit number
Motor[*x*].EncLossLevel = 1 // High-true fault

Note that instead of the **Gate3[*i*]** structure name, it is also possible to use the alias name for the particular board: **Acc24E3[*i*]** or **PowerBrick[*i*]**.

Analog Sinusoidal Encoders and Resolvers

For analog sinusoidal encoders and resolvers connected to an ACC-24E3 UMAC axis-interface board with a PMAC3-style DSPGATE3 ASIC, or to a Power Brick control board, the following settings should be used:

Motor[*x*].pEncLoss = **Gate3[*i*].Chan[*j*].SosError.a** // Sum-of-squares error register
Motor[*x*].EncLossBit = 31 // Sum-of-squares error bit number
Motor[*x*].EncLossLevel = 1 // High-true fault

Note that instead of the **Gate3[*i*]** structure name, it is also possible to use the alias name for the particular board: **Acc24E3[*i*]** or **PowerBrick[*i*]**.

There is not a dedicated loss-error bit for analog sinusoidal encoders connected to an ACC-51E UMAC board, or for resolvers connected to an ACC-58E UMAC board.

Serial Encoders

For serial encoders connected to an ACC-24E3 UMAC axis-interface board or to a Power Brick control board with a PMAC3-style DSPGATE3 ASIC, the following settings should be used for encoder protocols with a "timeout error" flag (EnDat, Hiperface, Sigma I, Sigma II/III/V, Tamagawa, Panasonic, Mitutoyo, and Kawasaki):

Motor[*x*].pEncLoss = **Gate3[*i*].Chan[*j*].SerialEncDataB.a** // Status & error register
Motor[*x*].EncLossBit = 31 // Timeout error bit number
Motor[*x*].EncLossLevel = 1 // High-true fault

Note that instead of the **Gate3[*i*]** structure name, it is also possible to use the alias name for the particular board: **Acc24E3[*i*]** or **PowerBrick[*i*]**.

The same settings are valid for an SSI encoder with parity checking to use the parity-error bit.

For serial encoders connected to an ACC-84E UMAC FPGA-based serial-encoder interface board, the following settings should be used for encoder protocols with a "timeout error" flag (presently implemented protocols are EnDat, Hiperface, Sigma I, Sigma II/III/V, Tamagawa, Panasonic, and BiSS):

Motor[*x*].pEncLoss = **Acc84E[*i*].Chan[*j*].SerialEncDataB.a** // Status & error register
Motor[*x*].EncLossBit = 31 // Timeout error bit number

Motor[*x*].EncLossLevel = 1 // High-true fault

Setting the Encoder Loss Limit Value

It is possible with these loss-detection circuits that a brief transient electrical condition can momentarily cause the circuit to indicate a sensor loss when none has actually occurred. For this reason, Power PMAC permits you to specify the number of occurrences of the loss detection before tripping on an error condition. This is done with saved setup element **Motor[*x*].EncLossLimit**.

Each real-time interrupt (RTI) period, Power PMAC will check for encoder loss on each motor with this functionality enabled. If the specified bit is in its "loss" state, Power PMAC will increment the status element **Motor[*x*].EncLossCount** by 1. If the specified bit is not in its "loss" state, Power PMAC will decrement this element by 1 (but never take it below 0).

If the value of **Motor[*x*].EncLossCount** ever exceeds that of **Motor[*x*].EncLossLimit**, an encoder-loss error will be generated. With **Motor[*x*].EncLossLimit** at its default value of 0, a single detection of the loss state will cause a trip.

The optimal setting of **Motor[*x*].EncLossLimit** must balance quick response to a true error while robustly avoiding any nuisance trips. Typically a setting of 3 or 4 will provide this balance.

Action on Encoder-Loss Error

When **Motor[*x*].EncLossCount** becomes greater than **Motor[*x*].EncLossLimit** and Power PMAC generates an "encoder-loss" error, it automatically "kills" the motor, putting it in open-loop mode with zero command output and amplifier disabled. It also aborts any motion program presently running in the motor's coordinate system. The status bit **Motor[*x*].EncLoss** is set to 1 to indicate this error.

If bit 0 (value 1) of **Motor[*x*].FaultMode** is set to its default value of 0, any other motors in the coordinate system will be "aborted" (decelerated to a closed-loop enabled stop according to **Motor[*x*].AbortTa** and **Motor[*x*].AbortTs**). If this bit is set to 1, any other motors in the coordinate system are also "killed". Motors in other coordinate systems are not affected in either case. Note that the action on an encoder-loss fault is identical to that for a fatal following error fault, amplifier fault, or I^2T fault.

Automatic Brake Control

Power PMAC provides the capability for automatic brake control on the motors it controls. The user can specify a digital output to be used to enable and disable a brake on the motor with configurable timing on the release and engagement of the brake as the motor is enabled and disabled. This is particularly useful for motors with a net load offset such as a gravity load on a vertical axis. Only a few saved setup elements must be configured to enable this functionality; no user algorithm is required.

Specifying the Brake Control Output

The brake control functionality is enabled by setting **Motor[*x*].pBrakeOut** to the address of the register containing the output bit. If this register (or the bit in the register) has an element name, the address can be specified using the "**.a**" suffix on the element name. For example:

Motor[4].pBrakeOut = **Acc24E3[1].Chan[0].OutFlagB.a**

Motor[5].pBrakeOut = **Acc68E[0].DataReg[5].a**

If there is no element name for the register, as with an ACC-11E UMAC I/O board, the address can be specified directly. For example:

Motor[6].pBrakeOut = **Sys.piom** + $A0000C

If **Motor[*x*].pBrakeOut** is left at its default value of 0, the automatic brake-control functionality for the motor is disabled.

Motor[*x*].BrakeOutBit specifies which bit in this 32-bit register is to be used for the output control. It is the location on the 32-bit data bus, not necessarily the offset from the lowest bit with an output. For example, I/O boards using the IOGATE IC have outputs only in bits 8 – 15 of the 32-bit bus, so a value from 8 to 15 should be used to specify one of these outputs.

Note that there is no software polarity control of the brake output bit. The bit is always set to 0 to engage the brake, and to 1 to release the brake. This is to encourage "fail-safe" implementations of the brake control, because a 0 typically sets a non-conducting output state, and most failure modes are non-conducting. In additions, output bits are forced to zero on controller reset (including watchdog timer trip) or shutdown.

Specifying the Brake Timing

Two saved setup elements control the timing for releasing and engaging the brake on the enabling and disabling of the motor. **Motor[*x*].BrakeOffDelay** specifies the delay in milliseconds from the time the motor is enabled (open-loop or closed-loop) and the specified brake-control output is set to 1 to release the brake. The purpose of the delay is to give the system enough time to ensure that proper control is established before brake release.

Motor[*x*].BrakeOnDelay specifies the delay in milliseconds from the time the specified brake-control output is set to 0 to engage the brake to the time the motor is disabled on a controlled (delayed) disabling. The purpose of the delay is to provide enough time for the brake to engage fully before servo control is removed. Delayed disabling is performed using the motor **`dkill`** command or the coordinate-system **`ddisable`** command; both of these can be given as on-line commands or buffered program commands.

There is no delay if standard disabling commands (**`k`**, **`kill`**, **`disable`**) are used, or if the motor is disabled due to a fault condition (fatal following error, amplifier fault, encoder loss). In these cases, the motor is killed at the same time the brake engagement is commanded.

Amplifier Enable and Fault Lines

The use of the amplifier-enable (AENAn) output and the amplifier-fault (FAULTn) input lines for each motor are important for safe operation. Without the use of the enable line, disabling the amplifier relies on precise zero offsets in Power PMAC's outputs and the amplifier's inputs. The amplifier-enable line used for the motor is specified by the address in saved setup element **Motor[*x*].pAmpEnable** (usually **Gate*n*[*i*].Chan[*j*].Ctrl.a**). It is usually part of the same ASIC channel as the other flags used for the motor.

The enable/disable polarity of the amplifier-enable line cannot be changed in software. From the software viewpoint, a 0 in the bit controlling the line means "disable", and a 1 means "enable".

Failures such as watchdog timer trip use hardware circuits to force the output to a "disable" (0) state (which is why software polarity control is not permitted).

Without the use of the fault line, Turbo PMAC may not know when an amplifier has shut down and may not take appropriate action. The amplifier-fault line used for the motor is specified by the address in **Motor[*x*].pAmpFault** (usually **Gate*n*[*i*].Chan[*j*].Status.a**). This is usually part of the same ASIC channel as the other flags used for the motor.

Saved setup element **Motor[*x*].AmpFaultBit** specifies which bit of the 32-bit register specified is read for the status of the amplifier input. This should be set to 23 when using a PMAC2-style "DSPGATE2" IC, as on an ACC-24E2x UMAC board, or to 23 when using the standard MACRO-ring protocol. It should be set to 7 when using a PMAC3-style "DSPGATE3" IC, as on an ACC-24E3 UMAC board.

The fault/no-fault polarity of the amplifier-fault input is determined by the saved setup element **Motor[*x*].AmpFaultLevel**. If set to the default value of 1, the input state that produces a 1 in the bit read by the processor is viewed as a fault condition. If set to 0, the input state that produces a 0 in the bit read by the processor is viewed as a fault condition.

On detecting an amplifier fault condition, the motor is automatically "killed" (open-loop, zero-output, amplifier disabled). If bit 0 (value 1) of **Motor[*x*].FaultMode** is set to the default value of 0, other motors in the coordinate system (even if they just have the "null" definition – **#x->0** – in that coordinate system) are automatically "aborted" (controlled deceleration to enabled, closed-loop stop). If bit 0 (value 1) of **Motor[*x*].FaultMode** is set to 1, other motors in the coordinate system are automatically "killed" as well.

Integrated Current (I^2T) Protection

Power PMAC can be set up to fault a motor if the time-integrated current levels exceed a certain threshold. This can protect the amplifier and/or motor from damage due to overheating. It integrates the square of current over time – commonly known as I^2T ("eye-squared-tee") protection. Power dissipation in a resistive element is proportional to the square of the current, and integrating this power dissipation over time provides a good estimate of the resulting heating.

Some amplifiers have their own internal integrated-current protection, but others do not. Power PMAC's integrated-current protection can be used in either case. It can be used with any amplifier for which Power PMAC computes current commands, whether or not Turbo PMAC also performs the commutation and/or digital current loop functions. If Power PMAC is closing the current loop for the motor, this function uses the measured current values; otherwise it uses the commanded current values. This protection is not suitable for use in systems where Power PMAC outputs a velocity command, either as an analog voltage or a pulse frequency.

Two saved setup elements control the functioning of the I^2T protection for each motor. **Motor[*x*].I2tSet** is the continuous current limit magnitude. It has the same units as the **Motor[*x*].MaxDac** instantaneous output limit, bits of a signed 16-bit output device (even if some resolution is used). Both have a maximum magnitude of 32,767, which is the size of Power PMAC's maximum possible output. Generally **Motor[*x*].I2tSet** will be 1/4 to 1/2 of the magnitude of **Motor[*x*].MaxDac**.

When the magnitude of the instantaneous current value is greater than **Motor[*x*].I2tSet**, the integrated current value will increase. When it is less than **Motor[*x*].I2tSet**, the integrated current value will decrease (until equal to zero).

Motor[*x*].I2tTrip is the integrated current limit parameter. If **Motor[*x*].I2tTrip** is set to 0, this function is disabled. If **Motor[*x*].I2tTrip** is greater than 0, Power PMAC will compare the integrated current value to **Motor[*x*].I2tTrip**. When the integrated current value exceeds this value, Power PMAC will fault this motor as if an amplifier fault had occurred. The offending motor is killed; if it was in a coordinate system running a motion program, that motion program aborted, and the motion of the other motors in the coordinate system is aborted.

Power PMAC's I^2T function works according to the following equation (using signed 16-bit output units):

$$Sum = Sum + \left[\left(\frac{I_q}{32768}\right)^2 + \left(\frac{I_d}{32768}\right)^2 - \left(\frac{Ixx57}{32768}\right)^2\right]\Delta t$$

where:

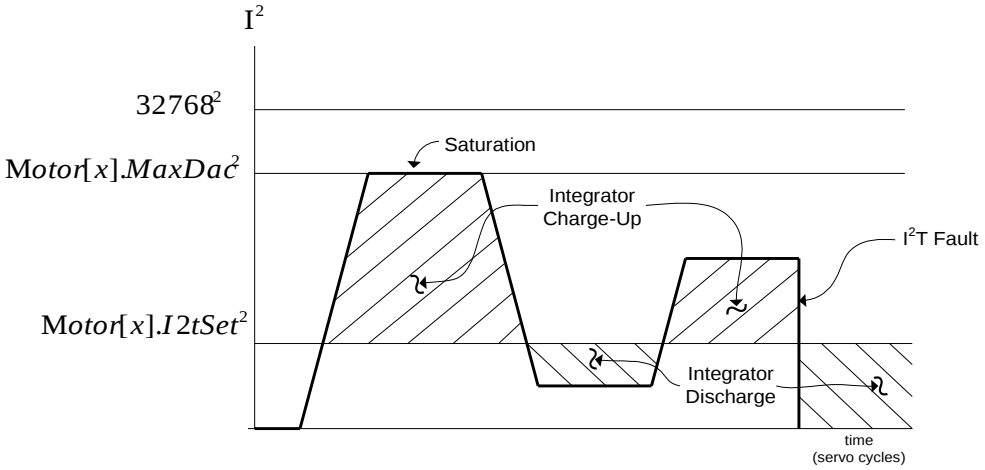
I_q (quadrature current) is the commanded torque-producing output of the PID filter in units of a 16-bit DAC;

I_d (direct current) is the magnetization current command as set by **Motor[*x*].IdCmd**. This is usually zero except when Power PMAC is doing vector control of induction motors.

Δt is the time since the last sample in seconds

If *Sum* exceeds **Motor[*x*].I2tTrip**, an integrated-current fault will occur. When instantaneous current levels are below **Motor[*x*].I2tSet**, *Sum* will decrease, but it will never go below zero.

Power PMAC I^2T Protection Feature



Example: With **Motor[*x*].MaxDac** = 30,000, continuous current limit **Motor[*x*].I2tSet** = 15,000 (half of maximum), magnetization current **Motor[*x*].IdCmd** = 0, and servo update time of 100 microseconds, the motor hits an obstruction, and the command output saturates at 30,000. The integrated-current protection function will calculate during this time:

$$Sum = Sum + \left[30{,}000^2 + 0^2 - 15{,}000^2\right]\Delta t$$
$$Sum = Sum + \left[9\times10^8 - 2.25\times10^8\right]10^{-4} = Sum + 6.75\times10^4$$

Sum will increase at a rate of 67,500 per servo cycle, or 675,000,000 per second. If you want the motor to trip after 2 seconds of this condition, you should set **Motor[*x*].I2tTrip** to 675,000,000 * 2, or 1,350,000,000 (1.35E9).

When an integrated-current fault occurs on a motor, Power PMAC reacts just as for an amplifier fault error. The offending motor is killed, and other motors in the coordinate system are aborted or killed, as determined by bit 0 of **Motor[*x*].FaultMode**. Power PMAC sets the amplifier fault motor status bit and a separate integrated-current fault motor status bit. Both bits are cleared when the motor is re-enabled.

Note: When Power PMAC is not commutating a motor with I^2T protection, make sure magnetization current parameter **Motor[*x*].IdCmd** is still set to 0. In this setup, **Motor[*x*].IdCmd** will not affect operation, but it will affect integrated-current calculations.

For maximum protection, Power PMAC performs the I^2T 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.

Velocity Limits

Power PMAC provides several limits on the velocities that can be commanded of axes and motors.

Programmed Vector Velocity Limit

Each coordinate system has a vector velocity limit in **Coord[*x*].MaxFeedrate**, expressed in axis units per time unit. (The time unit is set by **Coord[*x*].FeedTime**, in milliseconds.) If a move is specified by vector velocity (linear or circle mode move specified with **F** instead of **tm**), the specified vector velocity (feedrate) is compared to this parameter. If the value is greater than the limit, the limit value is used for the move instead.

Programmed Motor Velocity Limit

Each motor has a programmable velocity limit in **Motor[*x*].MaxSpeed**, expressed in motor units per millisecond. This limit has several functions. First, it serves as the commanded velocity for the motor in rapid-mode moves if the **Motor[*x*].RapidSpeedSel** is set to the default value of 0.

Second, for linear-mode moves (with or without move segmentation enabled), **Motor[*x*].MaxSpeed** serves as the maximum velocity permitted, calculated on a move-by-move basis. If the commanded velocity requested of a motor for a move exceeds the limit for the motor, the move is slowed so that the velocity limit is not exceeded. In a multi-axis programmed move,

all axes in the coordinate system are slowed proportionally so that no change in path occurs and coordination is maintained.

In addition, for linear, circle and PVT-mode moves executed with segmentation enabled (**Coord[*x*].SegMoveTime** > 0) and the special lookahead buffer active (lookahead buffer defined, **Coord[*x*].LHDistance** > 0), it serves as the maximum velocity for each segment of the motion. This can be particularly valuable for non-Cartesian coordinate systems defined with Power PMAC's kinematic equations; very high motor velocities can inadvertently be commanded near "singularities". The lookahead algorithm can detect these problems beforehand, and slow the motion down along the path into the problem point, observing the **Motor[*x*].InvAmax** motor acceleration limits for all axes in the coordinate system.

Velocities are compared to these limits assuming the % value for the coordinate system is 100 (real time), as the % value is used after this limit check. This means that other % values will result in different effective limits. However, for segmented moves, this limit check occurs after the segmentation override value **Coord[*x*].SegOverride** is used, so the effective limit remains the same regardless of what segmentation override value is used.

Position-Following Velocity Limit

Each motor can specify the maximum velocity magnitude that can result from the position-following function. If **Motor[*x*].MasterMaxSpeed** is greater than zero, it specifies this limit in motor units per servo cycle. If the speed of the master and the following ratio in **Motor[*x*].MasterPosSf** request a higher speed in any servo cycle, the resulting speed will be limited to this magnitude. Note that if this following is superimposed on programmed moves, only the component of speed from the following is limited by this parameter; total speed could be greater. For more detail on this feature, refer to position-following section of the chapter *Synchronizing Power PMAC to External Events.*

Acceleration Limits

Power PMAC provides several limits on the accelerations that can be commanded of axes and motors.

Programmed Vector Acceleration Limits

Each coordinate system has two vector acceleration limits that act at move computation time. They are intended for path-based applications, and are calculated in the plane in XYZ Cartesian space that is defined by the normal command. By default, the XY-plane is used.

Coord[*x*].MaxCirAccel limits the V^2/R centripetal acceleration from a programmed circle-mode move. If the programmed velocity would produce a centripetal acceleration higher than this limit, the velocity is automatically slowed at the move computation time so that this limit is not violated. This is done independently of any motor acceleration limiting.

Coord[*x*].CornerAccel specifies the acceleration in blending between any two linear and/or circle-mode moves, calculated based on the corner angle in the specified plane and the programmed speed of the moves. It calculates the blending time necessary to achieve this acceleration without exceeding it, ignoring the value in **Coord[*x*].Ta**, but will not use a time less than **Coord[*x*].Td**.

Programmed Motor Acceleration Limit

Each motor has a programmable acceleration limit set by **Motor[*x*].InvAmax**, expressed in milliseconds^2 per motor unit. As the name implies, this is the inverse of the maximum rate of acceleration. This limit has multiple functions.

First, for simple linear-mode moves with move segmentation disabled (**Coord[*x*].SegMoveTime** = 0), **Motor[*x*].InvAmax** specifies as the maximum acceleration permitted, calculated on a move-by-move basis. If the commanded acceleration requested of a motor for a move by the change in velocity and the acceleration times exceeds the limit for the motor, the acceleration times are extended so that the acceleration limit is not exceeded. In a multi-axis programmed move, the times for all axes in the coordinate system are identically extended so that full coordination is maintained.

Note that in blending linear-mode moves, the acceleration time occurs half in the incoming move and half in the outgoing move. Extending the acceleration time of a blend therefore causes the blend to occupy more time in both the incoming and outgoing moves. It is not possible to extend the acceleration time past the start of the constant-speed portion of the incoming move. In this mode of operation, if observing the acceleration limit requires that the blend time be extended past this point, the blend time will only be extended to this point, and the acceleration limit will be violated. (If your application requires more sophisticated acceleration limiting than this, you will need to use the special buffered lookahead function that can spread acceleration over multiple moves. This is discussed below.)

Second, for linear, circle, and PVT-mode moves executed with segmentation enabled (**Coord[*x*].SegMoveTime** > 0) and the special lookahead buffer active (lookahead buffer defined, **Coord[*x*].LHDistance** > 0), it serves as the maximum acceleration for each segment of the motion. The lookahead algorithm can detect these problems beforehand, and slow the motion down along the path into the problem point, observing the **Motor[*x*].InvAmax** motor acceleration limits for all axes in the coordinate system.

Accelerations are compared to these limits assuming the % value for the coordinate system is 100 (real time), as the % value is used after this limit check. This means that other % values will result in different effective limits. However, for segmented moves, this limit check occurs after the segmentation override value **Coord[*x*].SegOverride** is used, so the effective limit remains the same regardless of what segmentation override value is used.

Position-Following Acceleration Limit

Each motor can specify the maximum acceleration magnitude that can result from the position-following function. If **Motor[*x*].MasterMaxAccel** is greater than zero, it specifies this limit in motor units per servo cycle per servo cycle. If the motion of the master and the following ratio in **Motor[*x*].MasterPosSf** request a higher acceleration magnitude in any servo cycle, the resulting acceleration will be limited to this magnitude. Note that if this following is superimposed on programmed moves, only the component of acceleration from the following is limited by this parameter; total acceleration could be greater. For more detail on this feature, refer to position-following section of the chapter *Synchronizing Power PMAC to External Events.*

Command Output Limits

Each motor has a programmable limit for the servo-loop command output magnitude set by **Motor[*x*].MaxDac**. If the servo algorithm computes a command output of a greater magnitude,

the magnitude of the output used is limited to this value. Despite the name of this element, it can be used even when the output device is not a D/A converter.

This limit is expressed in units of a signed 16-bit output (even if the actual output device has a different resolution). Full range is thus 32,767 (2^{15}-1). If **Motor[*x*].MaxDac** is set to 32,767, it performs no limiting function. It is used regardless of the servo algorithm selected.

The limit is used both in the case where the servo-loop output is written directly to the output device (i.e. no Power PMAC commutation for the motor) and the case where the servo-loop output serves as the torque (quadrature) command input to the commutation algorithm. For torque-mode, sinewave-mode, and direct-PWM amplifiers, this limit serves as a torque, or current-magnitude limit. For velocity-mode or pulse-and-direction amplifiers, it serves as a velocity limit.

In servo cycles where the limit is active, reducing the command output of the servo loop, Power PMAC's anti-windup protection activates (for the built-in servo algorithms) to prevent oscillation when coming out of the limiting condition.