Navigator™ Motion Processor

User's Guide

P M D

Performance Motion Devices, Inc.
12 Waltham St.
Lexington, MA 02421

Revision 1.4, December 2001

Warranty

PMD warrants performance of its products to the specifications applicable at the time of sale in accordance with PMD's standard warranty. Testing and other quality control techniques are utilized to the extent PMD deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Performance Motion Devices, Inc. (PMD) reserves the right to make changes to its products or to discontinue any product or service without notice, and advises customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

Safety Notice

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage. Products are not designed, authorized, or warranted to be suitable for use in life support devices or systems or other critical applications. Inclusion of PMD products in such applications is understood to be fully at the customer's risk.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent procedural hazards.

Disclaimer

PMD assumes no liability for applications assistance or customer product design. PMD does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of PMD covering or relating to any combination, machine, or process in which such products or services might be or are used. PMD's publication of information regarding any third party's products or services does not constitute PMD's approval, warranty or endorsement thereof.

Related Documents

Navigator Motion Processor User's Guide (MC2000UG)

How to set up and use all members of the Navigator Motion Processor family.

Navigator Motion Processor Programmer's Reference (MC2000PR)

Descriptions of all Navigator Motion Processor commands, with coding syntax and examples, listed alphabetically for quick reference.

Navigator Motion Processor Technical Specifications

Five booklets containing physical and electrical characteristics, timing diagrams, pinouts, and pin descriptions of each series:

MC2100 Series, for brushed servo motion control (MC2100TS);
MC2300 Series, for brushless servo motion control (MC2300TS);
MC2400 Series, for microstepping motion control (MC2400TS);
MC2500 Series, for stepping motion control (MC2500TS);
MC2800 Series, for brushed servo and brushless servo motion control (MC2800TS).

Navigator Motion Processor Developer's Kit Manual (DK2000M)

How to install and configure the DK2000 developer's kit PC board.

Table of Contents

1 The Navigator Family

	MC2100 Series	MC2300 Series	MC2400 Series	MC2500 Series	MC2800 Series
# of axes	4, 2, or 1	4, 2 or 1	4, 2 or 1	4, 2, or 1	4 or 2
Motor type supported	Brushed servo	Brushless servo	Stepping	Stepping	Brushed servo + brushless servo
Output format	Brushed servo (single phase)	Commutated (6-step or sinusoidal)	Microstepping	Pulse and direction	Brushed servo (single phase) + commutated (6-step sinusoidal)
Incremental encoder input	√	√	√	√	√
Parallel word device input	√	√	√	√	√
Parallel communication	√	√	√	√	√
Serial communication	√	√	√	√	√
Diagnostic port	√	√	√	√	√
S-curve profiling	√	√	√	√	√
Electronic gearing	√	√	√	√	√
On-the-fly changes	√	√	√	√	√
Directional limit switches	√	√	√	√	√
Programmable bit output	√	√	√	√	√
Software-invertable signals	√	√	√	√	√
PID servo control	√	√	-	-	√
Feedforward (accel & vel)	√	√	-	-	√
Derivative sampling time	√	√	-	-	√
Data trace/diagnostics	√	√	√	√	√
PWM output	√	√	√	-	√
Motion error detection	√	√	√ (with encoder)	√ (with encoder)	√
Axis settled indicator	√	√	√ (with encoder)	√ (with encoder)	√
DAC-compatible output	√	√	√	-	√
Pulse & direction output	-	-	-	√	-
Index & Home signals	√	√	√	√	√
Position capture	√	√	√	√	√
Analog input	√	√	√	√	√
User-defined I/O	√	√	√	√	√
External RAM support	√	√	√	√	√
Multi-chip synchronization	√ (21x3)	√ (23x3)	√ (24x3)		√ (28x3)
Chipset part numbers	MC2140 (4 axes) MC2120 (2 axes) MC2110 (1 axis)	MC2340 (4 axes) MC2320 (2 axes) MC2310 (1 axis)	MC2440 (4 axes) MC2420 (2 axes) MC2410 (1 axis)	MC2540 (4 axes) MC2520 (2 axes) MC2510 (1 axis)	MC2840 (4 axes) MC2820 (2 axes)
Developer's Kit p/n's:	DK2100	DK2300	DK2400	DK2500	DK2800

Introduction

This manual provides a User's Guide for the Navigator Family of Motion Processors from PMD including the MC2100 Series (brushed servo), MC2300 Series (brushless servo), MC2400 Series (microstepping), MC2500 Series (stepping), and MC2800 Series (brushed and brushless servo) chipsets.

Each of these devices is a complete chip-based motion processor, providing trajectory generation and related motion control functions. Depending on the type of motor controlled they provide servo loop closure, on-board commutation for brushless motors, and high speed pulse and direction outputs. Together these products provide a software-compatible family of dedicated motion processors that can handle a large variety of system configurations.

Each of these chips utilizes a similar architecture, consisting of a high-speed DSP (Digital Signal Processor) computation unit, along with an ASIC (Application Specific Integrated Circuit). The computation unit contains special on-board hardware that makes it well suited for the task of motion control.

Along with similar hardware architecture these chips also share most software commands, so that software written for one chipset may be re-used with another, even though the type of motor may be different.

Each chipset consists of two PQFP (Plastic Quad Flat Pack) ICs: a 100-pin Input/Output (I/O) chip, and a 132-pin Command Processor (CP) chip.

The four different series in the Navigator family are designed for a particular type of motor or control scheme. Here is a summary description of each series.

Family Summary

MC2100 Series (MC2140, MC2120, MC2110) – This series outputs motor commands in either Sign/Magnitude PWM or DAC-compatible format for use with brushed servo motors, or with brushless servo motors having external commutation.

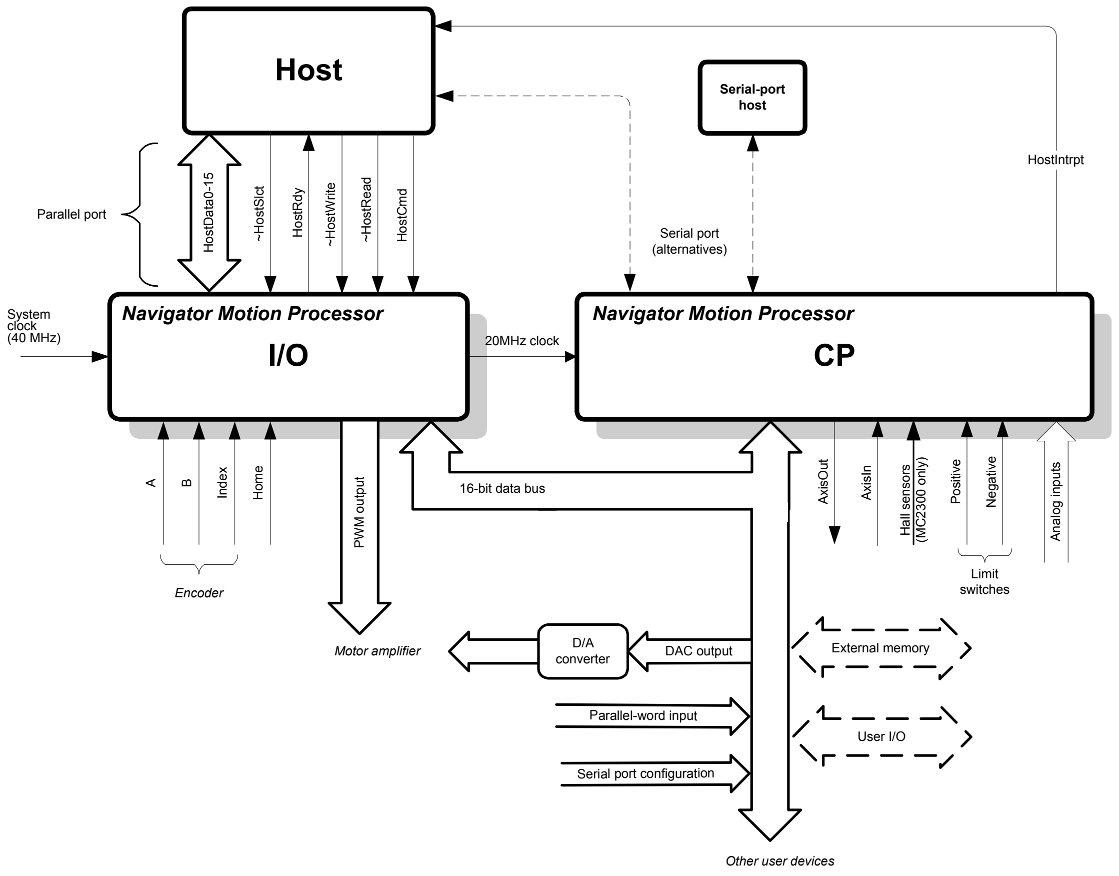
MC2300 Series (MC2340, MC2320, MC2310) – This series outputs sinusoidally commutated motor signals appropriate for driving brushless motors. Depending on the motor type, the output is a two-phase or three-phase signal in either PWM or DAC-compatible format.

MC2400 Series (MC2440, MC2420, MC2410) – This series provides microstepping signals for stepping motors. Two phased signals per axis are generated in either PWM or DAC-compatible format.

MC2500 Series (MC2540, MC2520, MC2510) – These chipsets provide high-speed pulse and direction signals for stepping motor systems.

MC2800 Series (MC2840, MC2820) – This series outputs sinusoidally or 6-step commutated motor signals appropriate for driving brushless servo motors as well as PWM or DAC- compatible outputs for driving brushed servo motors.

2 System Overview



The above figure shows a block diagram of the Navigator motion processors.

Each axis inputs the actual location of the axis using either incremental encoder signals or a parallel-word input device such as an absolute encoder, Analog to Digital converter, resolver, or laser interferometer. If incremental signals are used, the incoming A and B quadrature data stream is digitally filtered, and then passed on to a high-speed up/down counter. Using the parallel-word interface, a direct binary-encoded position of up to 16 bits is read by the chipset. Regardless of the encoder input method, this position information is then used to maintain a 32-bit actual axis position counter.

The CP chip contains a trajectory generator that calculates a new desired position at each cycle time interval based on the profile modes and profile parameters programmed by the host, as well as the current state of the system. The cycle time is the rate at which major system parameters are updated such as trajectory, servo compensation (if using MC2100 Series, MC2300 Series or MC2800), and some other chipset functions.

For the servo control chipsets (MC2100, MC2300 and MC2800) the output of the trajectory generator is combined with the actual encoder position to calculate a 32-bit position error, which is passed through a PID filter. The resultant value is then output by the chipset to an external amplifier using either PWM or DAC signals. If the MC2300 (brushless servo) or MC2800 (brushed and brushless servo) chipset is used then the output signals are commutated, meaning they are combined with information about the motor phase angle to distribute the desired motor torque to 2 or 3 phased output commands. With the MC2800 chipset (brushed servo) commutation is not performed, and the single-phase motor command is output directly.

For the stepping motor chipsets (MC2400 and MC2500) the output of the trajectory generator is converted to either microstepping signals or pulse and direction signals and then output accordingly. Microstepping signals are output in either PWM or DAC format.

Communication to/from the Navigator motion processors is accomplished using a parallel-bus interface and/or an asynchronous serial port. If parallel-bus communication is used there is a further choice of 8-bit wide transfers or 16-bit wide transfers. This allows a range of microprocessors and data buses to be interfaced to. If serial communications are used then the user selects parameters such as baud rate, number of stop/start bits, and the transfer protocol. The transfer protocol can be either point-to-point (appropriate for single chipset systems) or multi-drop (appropriate for serial communications to multiple chipsets).

Communication to/from the Navigator chipsets occurs using short commands sent or received as a sequence of bytes and words. These packets contain an instruction code word that tells the motion processor what operation is being requested. It might also contain data sent to the motion processor, or data received from the motion processor.

These commands are sent by a host microprocessor or host computer that executes a supervisor program, thereby providing overall system control. The Navigator motion processors are designed to function as the motion engine, managing high speed dedicated motion functions such as trajectory generation, safety monitoring, etc. while the host software program provides the overall motion sequences.

Other major functions of the Navigator chipsets include:

Breakpoints - Breakpoints allow various signals or parameters to be monitored and compared against user programmed conditions. Breakpoints can be programmed to automatically adjust the chipset behavior when the condition is satisfied. This is useful for functions such as "change the trajectory velocity when a signal goes high."

Diagnostic parameter capture - Diagnostic parameter capture allows up to four chipset parameters to be stored automatically in an external RAM chip for later examination by the host. This facility makes it easy to generate very accurate graphs of servo performance, trajectory information, etc., without any real time data collection involvement by the host.

Motion error, tracking window, and settle window - these functions perform automatic monitoring of the position error (the difference between the desired position and the actual encoder position). They are useful for performing functions such as "stop the axis if a particular position error value is exceeded" or "define the motion as being complete when the axis is within a programmed position error for a programmed amount of time."

Limit switches - Allows the axis to be automatically stopped if the motion travel is beyond the legal range.

Other chipset features include analog signal input, software invertable digital signals, and user-defined I/O decoding, to name a few.

In the following sections each of these chipset functions will be discussed and explained. Most chipset functions are common across all Navigator motion processors. However, some sections of this manual describe features that apply to particular chipsets only. For example, servo filtering applies to the MC2100 Series, MC2300 Series and MC2800 Series only. These sections are clearly marked as such.

3 Trajectory Generation

3.1 Trajectories, profiles, and parameters

The trajectory generator performs calculations to determine the instantaneous position, velocity and acceleration of each axis at any given moment in time. These values are called the *commanded* values. During a motion profile, some or all of these parameters will change continuously. Once the move is complete these parameters will stay at the same value until a new move is started.

To query the instantaneous commanded profile values the commands GetCommandedPosition, GetCommandedVelocity, and GetCommandedAcceleration are used.

Throughout this manual various command mnemonics will be shown to clarify chipset usage or provide specific examples. See the *Navigator Programmers Reference* for more information on host commands, nomenclature, and syntax.

The specific profile that is created by the Navigator depends on several factors including the presently selected profile mode, the presently selected profile parameters, and other system conditions such as whether a motion stop has been requested. Four trajectory profile modes are supported: S-curve point to point, Trapezoidal point to point, Velocity contouring and Electronic gearing. The operation of these profile modes will be explained in detail in subsequent sections. The command used to select the profile mode is SetProfileMode. The command GetProfileMode retrieves the programmed profile mode.

The profile mode may be programmed independently for each axis. For example axis #1 may be in trapezoidal mode while axis #2 is in S-curve point to point.

With one exception, Navigator motion processors can switch from one profile to another while an axis is in motion. The exception: when switching to the S-curve point-to-point profile from any other profile, the axis must be at rest.

3.1.1 Trajectory parameter representation

The Navigator Motion Processor sends and receives trajectory parameters using a fixed point representation. In other words, a fixed number of bits are used to represent the integer portion of a real number, and a fixed number of bits are used to represent the fractional component of a real number. The chipset uses three formats.

Format	Word size	Range	Description
32.0	32 bits	- 2,147,483,648 to +2,147,483,647	Unity scaling. This format uses an integer only representation of the number.
16.16	32 bits	-32,768 to 32,767 + 65,535/65,536	Uses $1/2^{16}$ scaling. The chipset expects a 32 bit number which has been scaled by a factor of 65,536. For example to specify a velocity of 2.75, 2.75 is multiplied by 65,536 and the result is sent to the chipset as a 32 bit integer (180,224 dec. or 2c000 hex.).
0.32	32 bits	- 2,147,483,648/4,294,967,296 to +2,147,483,647/4,294,967,296	Uses $1/2^{32}$ scaling. The chipset expects a 32 bit number which has been scaled by a factor of 4,294,967,296 (2^{32}). For example to specify a value of .0075, .0075 is multiplied by 4,294,967,296 and the result is sent to the chipset as a 32 bit integer (32,212,256 dec. or 1eb8520 hex).

3.2 Trapezoidal point-to-point profile

The following table summarizes the host specified profile parameters for the trapezoidal point to point profile mode:

Profile Parameter	Format	Word size	Range
Position	32.0	32 bits	- 2,147,483,648 *to* 2,147,483,647 counts.
Velocity	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle.
Acceleration	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle^2.
Deceleration	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle^2.

The host instructions SetPosition, SetVelocity, SetAcceleration, and SetDeceleration load these values. The commands GetPosition, GetVelocity, GetAcceleration, and GetDeceleration retrieve the programmed values.

For this profile, the host specifies an initial acceleration and deceleration, a velocity, and a destination position. The profile gets its name from the resulting curve (Figure 3.2-1): the axis accelerates linearly (at the programmed acceleration value) until it reaches the programmed velocity. It continues in motion at that velocity, then decelerates linearly (using the deceleration value) until it stops at the specified position.

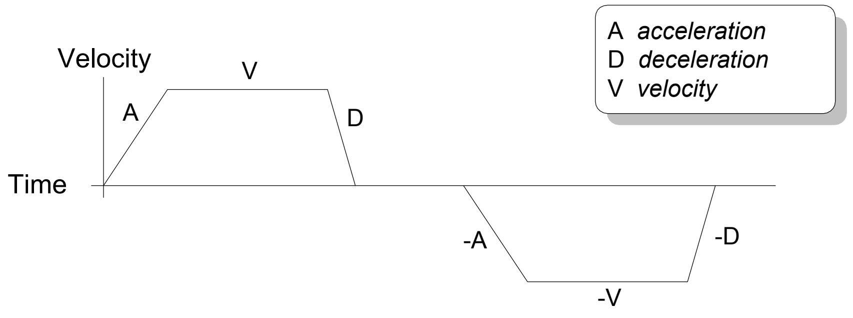


Figure 3.2-1. Simple trapezoidal point-to-point profiles

If deceleration must begin before the axis reaches the programmed velocity, the profile will have no constant velocity portion, and the trapezoid becomes a triangle (Figure 3.2-2).

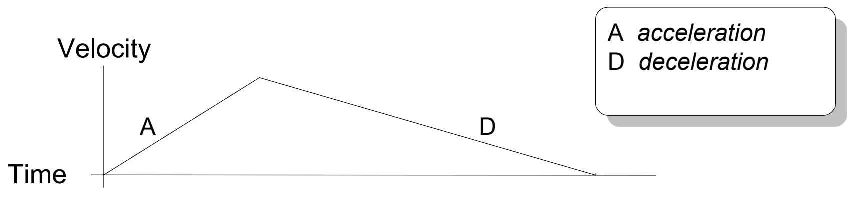


Figure 3.2-2. Simple trapezoidal point-to-point profiles

The slopes of the acceleration and deceleration segments may be symmetric (if acceleration equals deceleration) or asymmetric (if acceleration is not equal to deceleration).

The acceleration parameter is always used at the start of the move. Thereafter, the acceleration value will be used when the absolute velocity is increasing, and deceleration will be used when the absolute velocity is decreasing. If no motion parameters are changed during the motion then the acceleration value will be used until the maximum velocity is reached, and the deceleration value will be used when ramping down to zero. When the direction is reversed, the deceleration parameter is used for acceleration to the target velocity.

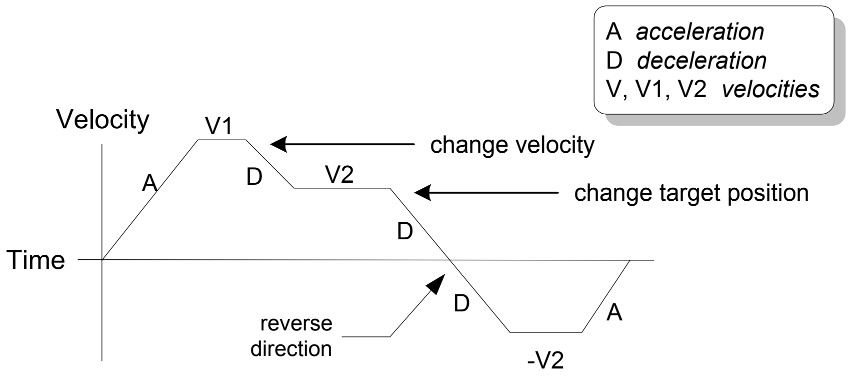


Figure 3.2-3. Complex trapezoidal profile, showing parameter changes

It is acceptable to change any of the profile parameters while the axis is moving in this profile mode. The profile generator will always attempt to remain within the legal bounds of motion specified by the parameters. If, during the motion, the destination position is changed in such a way that an overshoot is unavoidable, the profile generator will decelerate until stopped, then reverse direction to move to the specified position. Note that since the direction of acceleration/deceleration is fixed at the start of the move, the deceleration value will be used when ramping up velocity for the final move to the destination position. This is shown in Figure 3.2-3.

If a deceleration value of 0 (zero) is programmed (or no value is programmed leaving the chipset's default value of zero), then the value specified for acceleration (SetAcceleration) will automatically be used to set the magnitude of deceleration.

3.3 S-curve point-to-point profile

The following table summarizes the host specified profile parameters for the S-Curve point to point profile mode:

Profile Parameter	Format	Word size	Range
Position	32.0	32 bits	- 2,147,483,648 *to* 2,147,483,647 counts.
Velocity	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle.
Acceleration	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle^{2}.
Deceleration	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle^{2}.
Jerk	0.32	32 bits	0 to 2,147,483,647/4,294,967,296 counts/cycle^{3}.

The host instructions SetPosition, SetVelocity, SetAcceleration, SetDeceleration, and SetJerk load these respective values. The commands GetPosition, GetVelocity, GetAcceleration, GetDeceleration, and GetJerk retrieve the programmed values.

In S-curve profile mode, the same value must be used for both acceleration and deceleration. Asymmetric profiles are not allowed.

The S-curve point-to-point profile adds a limit to the rate of change of acceleration to the basic trapezoidal curve. A new parameter (*jerk*) is added which specifies the maximum change in acceleration in a single cycle.

In this profile mode, the acceleration gradually increases from 0 to the programmed acceleration value, then the acceleration decreases at the same rate until it reaches 0 again at the programmed velocity. The same sequence in reverse brings the axis to a stop at the programmed destination position.

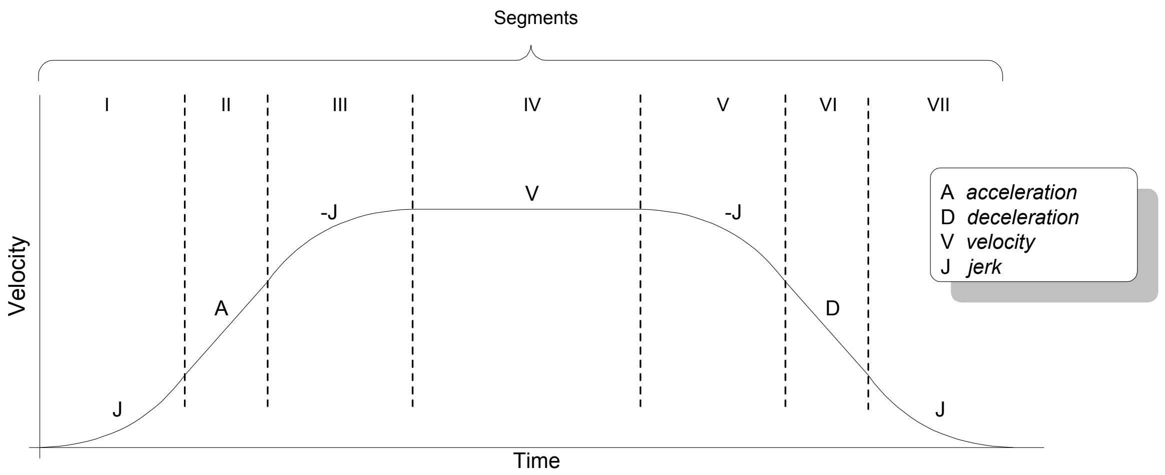


Figure 3.3-1 S-curve profile

Figure 3.3-1 shows a typical S-curve profile. In Segment I, the S-curve profile drives the axis at the specified jerk (J) until the maximum acceleration (A) is reached. The axis continues to accelerate linearly (jerk = 0) through Segment II. The profile then applies the negative value of the jerk to reduce acceleration to 0 during Segment III. The axis is now at maximum velocity (V), at which it continues through Segment IV. The profile will then decelerate in a manner similar to the acceleration stage, using the jerk value first to reach the maximum deceleration (D), and then to bring the axis to a halt at the destination.

An S-curve profile might not contain all of the segments shown in Figure 3.3-2. For example, if the maximum acceleration cannot be reached before the "halfway" point to or from the velocity, the profile would not contain a Segment II or a Segment VI. Such a profile is shown in Figure 3.3-2.

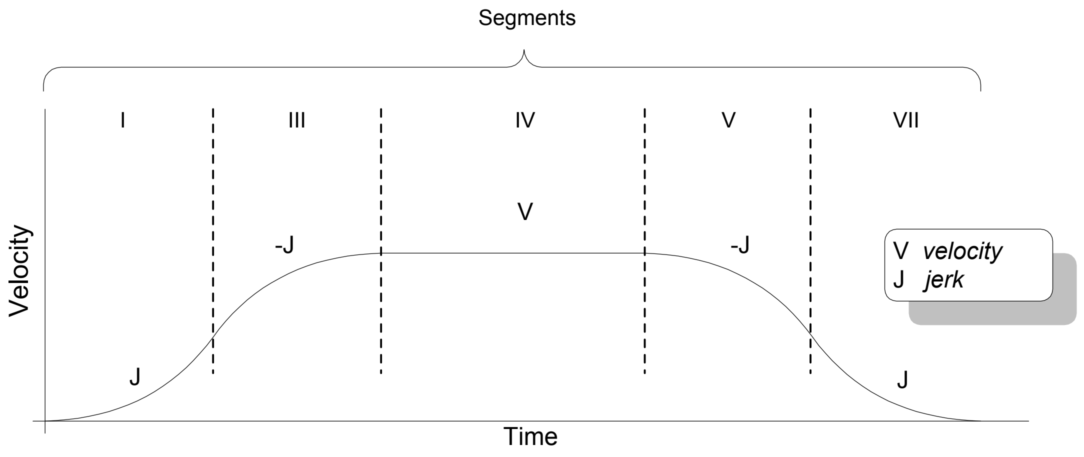


Figure 3.3-2. S-curve that doesn't reach maximum acceleration

Similarly, if the position is specified such that velocity is not reached, there will be no Segment IV, as shown in Figure 3.3-3. (There may also be no Segment II or Segment VI, depending on where the profile is "truncated.")

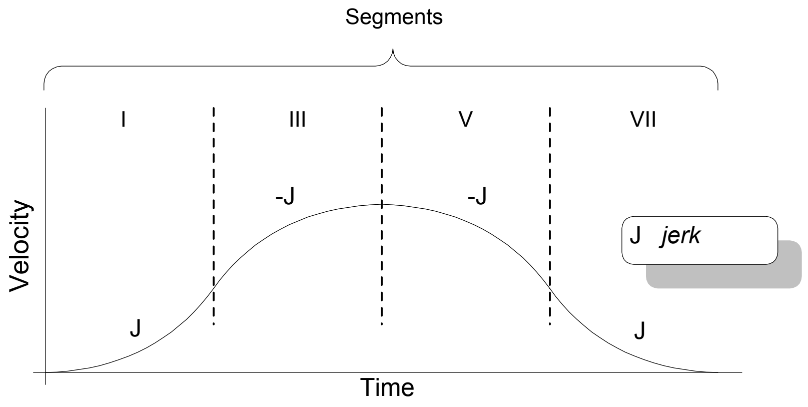


Figure 3.3-3. S-curve with no maximum-velocity segment

Unlike the trapezoidal profile mode, the S-curve profile mode does not support changes to any of the profile parameters while the axis is in motion.

An axis may not be switched into S-curve profile mode while the axis is in motion. It is however legal to switch from S-curve mode to any other profile mode while in motion.

3.4 Velocity-contouring profile

The following table summarizes the host specified profile parameters for the velocity contouring profile mode:

Profile Parameter	Format	Word size	Range
Velocity	16.16	32 bits	-32,768 to 32,767 + 65,535/65,536 counts/cycle.
Acceleration	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle$^{2.}$
Deceleration	16.16	32 bits	0 to 32,767 + 65,535/65,536 counts/cycle$^{2.}$

The host instructions SetVelocity, SetAcceleration, and SetDeceleration load these respective values. The commands GetVelocity, GetAcceleration, and GetDeceleration retrieve the programmed values.

Unlike the trapezoidal and S-curve profile modes where the destination position determines the direction of initial travel, in the velocity contouring profile mode the sign of the velocity parameter determines the initial direction of motion. Therefore the velocity value that is sent to the chipset can have positive values (for positive direction motion) or negative values (for negative direction motion).

In this profile, no destination position is specified. The motion is controlled entirely by changing the acceleration, velocity, and deceleration parameters while the profile is being executed.

In velocity contouring profile mode axis motion is not bounded by a destination. It is the host's responsibility to provide acceleration, deceleration, and velocity values which result in safe motion within acceptable position limits.

The trajectory is executed by continuously accelerating the axis at the specified rate until the velocity is reached. The axis starts decelerating when a new velocity is specified which has a smaller value (in magnitude) than the present velocity, or has a sign that is opposite to the present direction of travel.

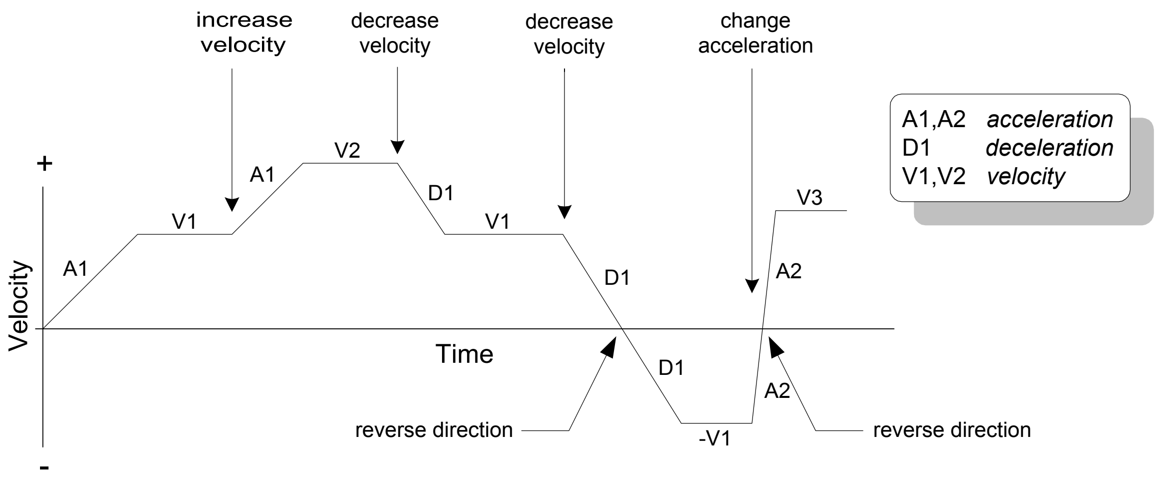


Figure 3.4-1. Velocity-contouring profile

A simple velocity-contouring profile looks just like a simple trapezoidal point-to-point profile, as shown in Figure 3.2-1.

Figure 3.4-1 illustrates a more complicated profile, in which both the velocity and the direction of motion change twice.

3.5 Electronic-gear profile

The following table summarizes the host specified profile parameters for the electronic gear profile mode:

Profile Parameter	Format	Word size	Range
Gear ratio	16.16	32 bits	-32,768 to 32,767 + 65,535/65,536 counts/cycle
Master axis #	-	4 bits	0 - 3*
Master source	-	1 bit	2 values; encoder or commanded (see below for details.)

*for two axis products 0 - 1. Single axis products do not support electronic gearing.

The host instructions SetGearRatio and SetGearMaster load these respective values. The commands GetGearRatio and GetGearMaster retrieve the programmed values.

In this profile, the host specifies three parameters. The first is the 'master' axis number which is the axis that will be the source of position information used to drive the 'slave' axis, which is the axis in gear mode. The second is the gear source, which is either actual (the encoder position of the master axis) or commanded (the commanded position of the master axis). The third is the gear ratio, which specifies the direction and ratio of master gear counts to slave counts.

Figure 3.5-1 shows the arrangement of encoders and motor drives in a typical electronic gearing application.

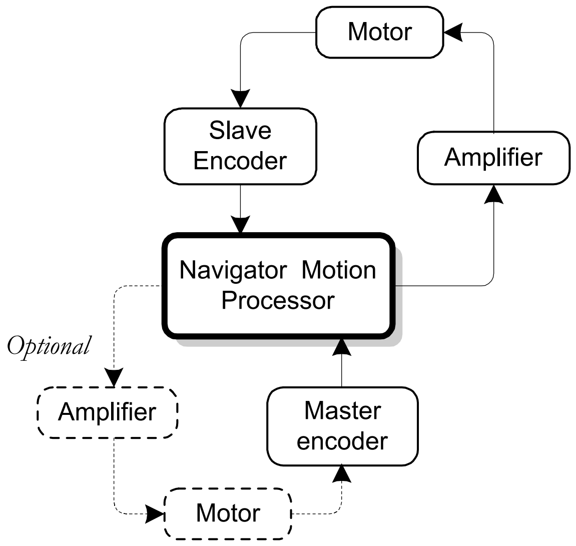


Figure 3.5-1. Axes set up for electronic-gear profile

A positive gear ratio value means that when the master axis actual or commanded position is increasing the slave commanded position will also increase. A negative gear ratio value has the opposite effect; increasing master position will result in decreasing slave axis commanded position.

For example, let us assume the slave axis is axis #0 (axes are counted 0, 1, 2, 3 for a four axis chipset) and the master axis are set to axis #3. Also assume the source will be 'actual' with a gear ratio of -

1/2. Then for each positive encoder count of axis 3, axis 0 commanded position will decrease in value by 1/2 count, and for each negative encoder count of axis 3, axis 0 commanded position will increase in value by 1/2 count.

The electronic gear profile requires two axes to be enabled. The single-axis motion processors, therefore, do not support electronic gearing in a useful way.

If the master axis source is set to 'actual', this axis need not have a physical motor attached to it. Frequently, it is used only for its encoder input, for example from a directly driven (open-loop) motor, or a manual control. It is possible, however, to drive a motor on the master axis by enabling the axis and applying a profile mode *other* than electronic gear to the axis. The effect of this arrangement is that both master and slave can be driven by the same profile, even though the slave can drive at a different ratio and in a different direction if desired. The master axis will operate the same whether or not it happens to be the master for some other geared axis. The 'optional' components shown in Figure 3.5-1 illustrate this arrangement. Such a configuration can be used to perform useful functions such as linear interpolation of two axes.

The gear-ratio parameter may be changed while the axis is in motion, but care should be taken to select ratios so that safe motion is maintained.

3.6 The SetStopMode command

Normally each of the above trajectory profile modes will execute the specified trajectory, within the specified parameter limits, until the profile conditions are satisfied. For example, for the point-to-point profile modes this means that the profile will move the axis until the final destination position has been reached, at which point the axis will have a velocity of zero.

In some cases however it is necessary to halt the trajectory manually, for safety reasons, or simply to achieve a particular desired profile. This can be accomplished using one of two methods: *abrupt stop*, or *smooth stop*.

To perform a stop the command SetStopMode is used. To retrieve the current stop mode the command GetStopMode is used.

Using the SetStopMode command to set the mode to AbruptStop instantaneously stops the profile by setting the target velocity of the designated axis to zero. This is, in effect, an emergency stop with instantaneous deceleration.

Setting the stop mode to SmoothStop brings the designated axis to a controlled stop, using the current deceleration parameter to reduce the velocity to zero.

In either mode, the target velocity is set to zero after the SetStopMode command is executed. Before any other motion can take place the velocity must then be re-set using the SetVelocity command.

AbruptStop **must be used with care. Sudden deceleration from a high velocity can damage equipment or cause injury.**

AbruptStop functions in all profiles. SmoothStop functions in all profiles except electronic-gear.

3.7 Motor Mode

All Navigator chipsets support a programmable motor mode that can enable and disable the profile generator, and for the servo chipsets (MC2100, MC2300 and MC2800) can set the chipsets to open loop mode or closed loop mode.

The command SetMotorMode determines the motor mode and the command GetMotorMode retrieves the current value of the motor mode.

If the motor mode is set to on then the trajectory generator is active. If the motor mode is set to off then the profile generator is disabled.

In addition, for the servo chipsets (MC2100 Series, MC2300 Series and MC2800 Series) if the motor mode is set off then the chipset enters open loop mode which means the servo filter is disabled and the motor command (the current output level requested by the chipset) is determined manually by the host using the command SetMotorCommand. If the motor mode is set on then the motor command is determined by the servo loop.

The most common use of the motor mode in anything other than the standard "on" state is after a motion error. In the case of a motion error (and if auto stop is enabled) then the chipset will set the motor mode off automatically, thereby placing it in a safe state where no further motion can occur until the host explicitly restores the motor mode to the on condition. For more information on motion errors see section 7.1.

For the servo chipsets (MC2100 Series, MC2300 Series and MC2800 Series) it may also be useful to set the motor mode to off for purposes of amplifier calibration.

3.8 Setting the cycle time

The chipset calculates all trajectory and servo information on a fixed, regular interval. This interval is known as the cycle time.

For each axis of the chipset that is enabled there is a minimum cycle time required for the chipset to function properly. The following table shows this:

Processor	**Cycle time per enabled axis**
MC2100	100 µsec
MC2300	150 µsec
MC2400	150 µsec
MC2500	100 µsec
MC2800	150 µsec

To calculate the total minimum cycle time for a given number of enabled axes, multiply the number of axes by the minimum cycle time. For example for a MC2100 with four axes enabled, the minimum loop time would be 4x100 = 400 µsec, for a cycle frequency of 2.5 kHz.

The cycle rate determines the trajectory update rate for all products as well as the servo loop calculation rate for the servo products (MC2100, MC2300 and MC2800). It does not however determine the commutation rate of the brushless servo products (MC2300 and MC2800).

An enabled axis receives its cycle "time slice" whether or not it is in motion, and whether or not the motor is on or off (SetMotorMode command). If cycle time is critical, it's possible to reclaim that time by disabling an unused axis and resetting the loop rate with the instruction SetSampleTime.

For example, using an MC2140, four axes are available, but if only 3 will be used in a specific application then the unused axis can be disabled using the command SetAxisMode and the new sample time of 300 µsec can be set using the SetSampleTime command. This improves the cycle frequency from 2.5 kHz to 3.333 kHz.

SetSampleTime may also be used to increase the cycle time to a value greater than the allowed minimum, if that should be necessary.

It is the responsibility of the host to make sure that the specified sample time is equal or larger than the specified minimums from the table above.

4 The Servo Loop

4.1 Overview

For the servo-based chipsets (MC2100, MC2300 and MC2800) a servo loop is used as part of the basic method of determining the motor command output. The function of the servo loop is to match as closely as possible the commanded position, which comes from the trajectory generator, and the actual motor position.

To accomplish this the profile generator *commanded value* is combined with the actual encoder position to create a position error, which is then passed through a digital PID-type servo filter. The scaled result of the filter calculation is the *motor command*, which is output as either a PWM signal to the motor amplifier, or a 16-bit input to a D/A Converter.

4.1.1 PID loop algorithm

The servo filter used with the MC2100, MC2300 and MC2800 chipsets is a proportional-integral-derivative (PID) algorithm, with velocity and acceleration feed-forward terms and an output scale factor. An integration limit provides an upper bound for the accumulated error. An optional bias value can be added to the filter calculation to produce the final motor output command. A limiting value for the filter output provides additional constraint. This limit is set using the command SetMotorLimit.

The PID+V_{ff}+A_{ff} formula, including the scale factor and bias terms, is as follows:

$$\text{Output}_n = \left[K_p E_n + K_d\left(E_k - E_{(k-1)}\right) + \sum_{j=0}^{n} E_j \times Ki / 256 + K_{vff}\left(CmdVel/\ 4\right) + K_{aff}\left(CmdAccel \times 8\right) \right]$$

$$\times \text{K}_{\text{out}}/65536 + \text{Bias}$$

where	E_n	are the accumulated error terms
	K_I	is the Integral Gain
	K_d	is the Derivative Gain
	K_p	is the Proportional Gain
	K_{aff}	is the Acceleration feed-forward
	K_{vff}	is the Velocity feed-forward
	Bias	is the DC motor offset
	K_{out}	is the scale factor for the output command.

All filter parameters, the motor output command limit, and the motor bias are programmable, so that the filter may be fine-tuned to any application. The parameter ranges, formats and interpretations are shown in the following table:

Term	Name	Representation & Range
I_{lim}	Integration Limit	unsigned 32 bits (0 to 2,147,483,647)
K_I	Integral Gain	unsigned 16 bits (0 to 32767)
K_d	Derivative Gain	unsigned 16 bits (0 to 32767)
K_p	Proportional Gain	unsigned 16 bits (0 to 32767)
K_{aff}	Acceleration feed-forward	unsigned 16 bits (0 to 32767)
K_{vff}	Velocity feed-forward	unsigned 16 bits (0 to 32767)
K_{out}	Output scale factor	unsigned 16 bits (0 to 32767)

Bias	DC motor offset	signed 16 bits (-32768 to 32,767)
	Motor command limit	unsigned 16 bits (0 to 32767)

The structure of the digital filter is shown in Figure 4.1-1.

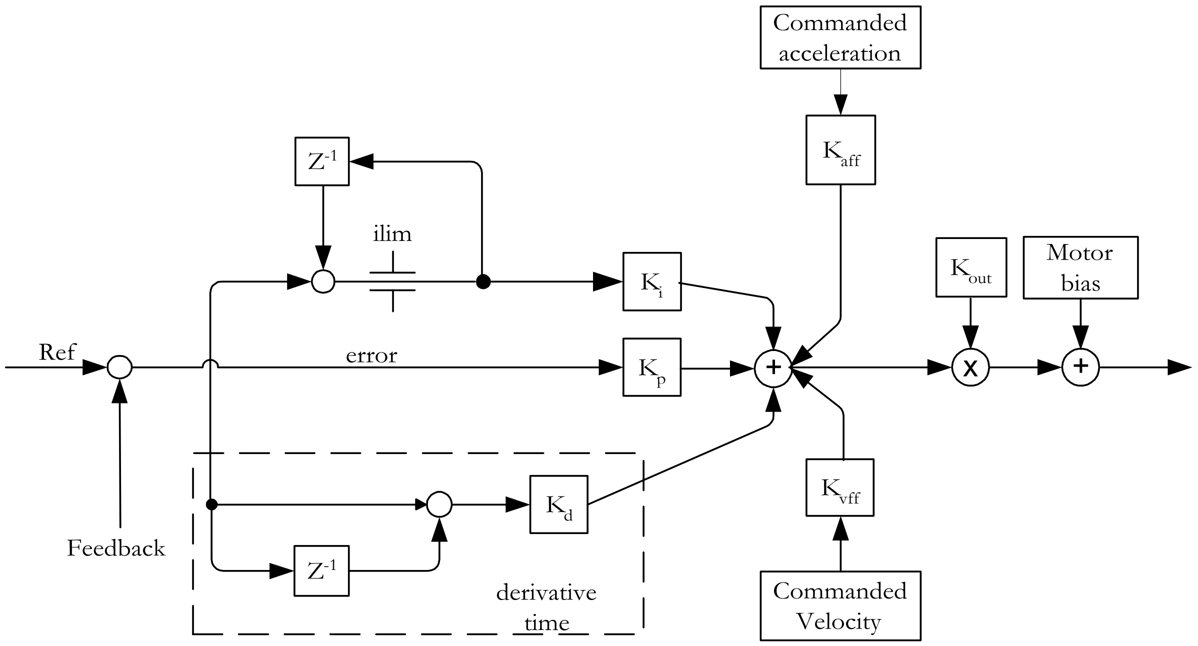


Figure 4.1-1. Digital Servo Filter

4.1.2 Motor bias

When an axis is subject to a net external force in one direction (such as a vertical axis pulled downward by gravity), the servo filter can compensate for it by adding a constant DC bias to the filter output. The bias value is set using the host instruction SetMotorBias. It can be read back using the command GetMotorBias.

4.1.3 Output scaling

The Kout parameter can be used to scale down the output of the PID filter in situations that require it. It does this by multiplying the filter result by Kout/65536. It has the effect of increasing the usable range of Kp, which is typically programmed in the 1 to 150 range when no output scaling is done. The Kout value is set using the host instruction SetKout. It can be read back using the command GetKout.

4.1.4 Output limit

The motor output limit prevents the filter output from exceeding a boundary magnitude in either direction. If the filter produces a value greater than the limit, the motor command takes the limiting value. The motor limit value is set using the host instruction SetMotorLimit. It can be read back using the command GetMotorLimit.

The motor limit applies only in closed-loop mode. It does not affect the motor command value set by the host in open-loop mode (see next section for more information on open and closed loop operations).

4.2 Closed-loop and open-loop control modes

In a previous section motor mode is discussed. For all Navigator chipsets setting the motor off (SetMotorMode Off) has the effect of disabling the trajectory generator.

In addition however, for the servo chipsets (MC2100, MC2300 and MC2800) turning the motor off, or having the motor be turned off automatically by the chipset via a motion error, places the chipset into what is known as 'open loop' mode. In open loop mode the servo filter does not operate and the motor command output value is set manually by the host using the command SetMotorCommand. With the motor 'on' the chipset is in 'closed loop' mode and the motor command value is controlled automatically by the servo filter.

Figure 4.2-1 shows the control flow for open and closed loop operation.

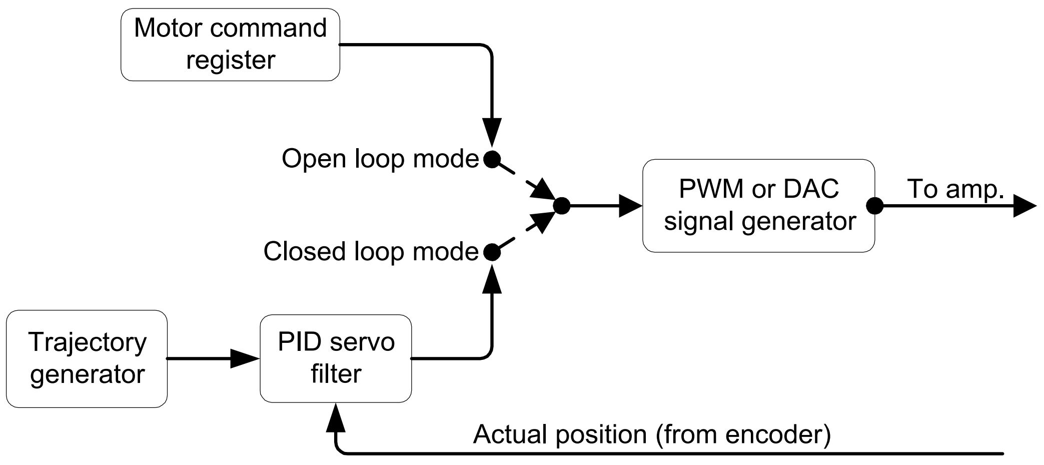


Figure 4.2-1. Motor control paths, closed- and open-loop modes

Closed-loop mode is the normal operating mode of the MC2100, MC2300 and MC2800. Open-loop mode is typically used when one or more axes require torque control only, or when the amplifier must be calibrated.

Limit switches do not function in open-loop mode.

4.2.1 Motor bias in open-loop mode

The motor bias applies at all times when operating in closed-loop mode. If the axis is switched to open-loop mode, the bias value continues to be output to the motor, to prevent the axis from suddenly lurching in the direction of the external force. Once the host issues a new motor command, however, its value supersedes the bias output, which no longer has any effect. As soon as the axis returns to closed-loop mode, the previous bias value is reinstated.

If the specified bias value does not properly compensate for the external force, the axis may move suddenly in one direction or another after a SetMotorMode *Off* instruction. It is the responsibility of the user to select a motor bias value that will maintain safe motion.

5 Parameter update and breakpoints

5.1 Parameter buffering

Various parameters must be specified to the chipset for an axis to be controlled correctly. In some cases it may be desired that a set of parameters take effect at the exact same time to facilitate precise synchronized motion.

To support this all profile parameters and several other types of parameters such as servo parameters (MC2100, MC2300 and MC2800 only) are loaded into the chipset using a buffered scheme. These buffered commands are loaded into an area of the chipset that does not affect the actual chipset behavior until a special event known as an Update occurs. An Update causes the buffered registers to be copied to the active registers, thereby causing the chipset to act on the new parameters.

For example the following command sequence loads a profile mode, position, velocity, and acceleration in but will not become active (take effect) until an Update is given:

```
SetProfileMode Axis1, trapezoidal     // set profile mode to trapezoidal for axis 1
SetPosition Axis1, 12345              // load a destination position for axis 1
SetVelocity Axis1, 223344             // load a velocity for axis 1
SetAcceleration Axis1, 1000           // load an acceleration for axis 1
```

After this sequence is completed the buffered registers for these parameters (including the profile mode itself) are loaded into the chipset but the trajectory generator module still operates on whatever the previous trajectory profile mode and parameters were. Only when an Update occurs will the trajectory profile mode actually be changed to trapezoidal and the specified parameters loaded into the trajectory generator, causing the trajectory generator to start the programmed motion.

5.1.1 Updates

There are three different ways that an Update can occur. They are listed below:

1) Update command - The simplest way is to give an Update command. This causes the parameters for the programmed axis to be updated immediately.

2) MultiUpdate command - The multiple axis instantaneous update, which is specified using the MultiUpdate command, causes multiple axes to be updated simultaneously. This can be useful when synchronized multi-axis profiling is desired. This command takes a 1-word argument that consists of a bit mask, with 1 bit assigned to each axis. Executing this command has the same effect as sending a set of Update commands to each of the individual axes selected in the MultiUpdate command mask.

3) Breakpoints - There is a very useful facility supported by the chipset that can be programmed to generate an Update command automatically when a pre-programmed condition becomes true. This feature is known as the breakpoint facility, and it is useful for performing operations such as "automatically change the velocity when a particular position is reached", or "stop the axis abruptly when a particular external signal goes active." Breakpoints are discussed in more detail in section 5.2.

Whichever Update method is used, at the time the update occurs, all of the buffered registers and commands will be copied to the active registers. However, depending on which calculations have already been performed in the servo loop, these values may not be used until the next cycle. Before the Update occurs, sending buffered commands will have no effect on the system behavior.

In addition to profile generation most servo parameter commands are buffered, and some other commands are buffered. Following is a complete list of buffered values and commands.

Buffered Commands

Trajectory	Servo & error tracking	Other
SetProfileMode	ClearPositionError	SetMotorCommand
SetAcceleration	SetIntegrationLimit	
SetJerk	SetKaff	
SetVelocity	SetKd	
SetPosition	SetKi	
SetDeceleration	SetKp	
SetGearRatio	SetKvff	
SetStartVelocity		
SetStopMode		

5.2 Breakpoints

Breakpoints are a convenient way of programming a chipset event upon some specific condition. Depending on the breakpoint instruction's arguments, a breakpoint can cause an update; an abrupt stop followed by an update; a smooth stop followed by an update; a motor-off followed by an update (more on this function in a later section); or no action whatsoever.

Each Navigator axis has two breakpoints that may be programmed for it. So two completely separate conditions may be monitored and acted upon. These two breakpoints are known as breakpoint 1 and breakpoint 2.

5.2.1 Defining a breakpoint, Overview

Each breakpoint has five components: the breakpoint axis, the source axis for the triggering event, the event itself, the action to be taken and the comparison value.

The *breakpoint axis* is the axis on which the specified action is to be taken.

The *source axis* is the axis on which the triggering event is located. It can be the same as or different than the breakpoint axis. Any number of breakpoints may use the same axis as a *source axis*.

The *trigger* is the event that causes the breakpoint.

The *action* is the sequence of operations executed by the chipset when the breakpoint is triggered. After a breakpoint is triggered the *action* is performed on the breakpoint axis.

The *comparison value* is used in conjunction with the action to define the breakpoint event.

Altogether these parameters provide great flexibility in setting breakpoint conditions. By combining these components, almost any event on any axis can cause a breakpoint.

The command used to send the breakpoint axis, the trigger, the source axis and the action is SetBreakpoint. To retrieve these same values the command GetBreakpoint is used. To set the comparison value the command SetBreakpointValue is used. This comparison value can be retrieved using the command GetBreakpointValue. For each of these commands the breakpoint number (1 or 2) must be specified.

The SetBreakpointValue command should always be sent before the SetBreakpoint command to set up a particular breakpoint.

5.2.2 Breakpoint triggers

The Navigator motion processors support the following breakpoint trigger conditions:

Trigger Condition	Level or Threshold	Description
GreaterOrEqualCommandedPosition	threshold	Is satisfied when the current commanded position is equal to or greater than the programmed compare value.
LesserOrEqualCommandedPosition	threshold	Is satisfied when the current commanded position is equal to or less than the programmed compare value.
GreaterOrEqualActualPosition	threshold	Is satisfied when the current actual position is equal to or greater than the programmed compare value.
LesserOrEqualActualPosition	threshold	Is satisfied when the current actual position is equal to or less than the programmed compare value.
CommandedPositionCrossed	threshold	Is satisfied when the current commanded position crosses (is equal to) the programmed compare value.
ActualPositionCrossed	threshold	Is satisfied when the current actual position crosses (is equal to) the programmed compare value.
Time	threshold	Is satisfied when the current chipset time (in number of cycles since power-up) is equal to the programmed compare value.
EventStatus	level	Is satisfied when the EventStatus register matches bit mask and high/low pattern in programmed compare value.
ActivityStatus	level	Is satisfied when the ActivityStatus register matches bit mask and high/low pattern in programmed compare value.
SignalStatus	level	Is satisfied when the SignalStatus register matches bit mask and high/low pattern set in programmed compare value.
none	-	Disables any previously set breakpoint.

If "none" is selected for the breakpoint trigger then this effectively means that that breakpoint is inactive. Only one of the above triggers can be selected at a given time. For a description of level triggered breakpoints refer to section 5.2.4.

5.2.3 Threshold-triggered breakpoints

Threshold triggered breakpoints use the value set using the SetBreakpointValue command as a single 32-bit threshold value to which a comparison is made. When the comparison is true, the breakpoint is triggered.

For example, if it is desired that the trigger occur when the commanded position is equal to or greater than 1,000,000, then the comparison value loaded using SetBreakpointValue would be 1,000,000, and the trigger selected would be PositiveCommandedPosition.

5.2.4 Level-triggered breakpoints

To set a level-triggered breakpoint, the host instruction supplies two 16-bit data words: a trigger mask and a sense mask. These masks are set using the SetBreakpointValue instruction. The high word of data passed with this command is the trigger mask value; the low word is the sense mask value.

The trigger mask determines which bits of the selected status register are enabled for the breakpoint. A 1 in any position of the trigger mask enables the corresponding status register bit to trigger a breakpoint, a 0 in the trigger mask disables the corresponding status register bit. If more then one bit is selected, then the breakpoint will be triggered when any selected bit enters the specified state.

The sense mask determines which state of the corresponding status bits causes a breakpoint. Any status bit that is in the same state (i.e. 1 or 0) as the corresponding sense bit is eligible to cause a breakpoint (assuming of course that it has been selected by the trigger mask).

For example, if the activity status register breakpoint has been selected, and the trigger mask contains the value 0402h and the sense mask contains the value 0002h, then the breakpoint will be triggered when bit 1 (the 'at max velocity' indicator) assumes the value 1, or bit 10 (the 'in motion' indicator) assumes the value 0.

5.2.5 Breakpoint actions

Once a breakpoint has been triggered, the chipset can be programmed to perform one of the following instruction sequences:

Action	Command Sequence Executed
None	No commands executed.
Update	Update *axis.*
Abrupt Stop	SetStopMode *axis,* AbruptStop Update *axis*
SmoothStop	SetStopMode *axis,* SmoothStop Update *axis*
MotorOff	SetMotorMode *axis*, Off Update *axis*

Regardless of the host's action, once a breakpoint condition has been satisfied, the Event Status bit corresponding to the breakpoint is set and the breakpoint is deactivated.

5.2.6 Breakpoint Examples

Here are a few examples to illustrate how breakpoints can be used.

Example #1 The host would like axis 1 to change velocity when the encoder position reaches a particular value. Breakpoint #1 should be used.

The following command sequence achieves this:

```
SetPosition Axis1, 123456               // Load destination
SetVelocity Axis1, 55555                // Load velocity
SetAcceleration Axis1, 500              // Load acceleration
SetDeceleration Axis1, 1000             // Load deceleration
Update Axis1                            // Make the move
SetVelocity Axis1, 111111               // Load a new velocity of 111,111 but do not send
                                        // an Update
SetBreakpointValue Axis1, 1, 100000     // Load 100,000 into the comparison register
                                        // for breakpoint 1
SetBreakpoint Axis1, 1, Axis1, Update, PositiveActualPosition
                                        // Specifiy a positive actual position breakpoint on axis 1
                                        // which will result in an Update when satisfied for
                                        // breakpoint 1
```

This sequence makes an initial move and loads a breakpoint after the first move has started. The breakpoint that is defined will result in the velocity being "updated" to 111,111 when the actual position reaches a value of 100,000. Therefore at 100,000 the axis will accelerate from a velocity of

55,555 to 111,111 at the acceleration value of 500. Note that any buffered registers that are not sent again will remain in the buffered registers. That is, when the breakpoint performs an Update the values for position, acceleration and deceleration are unchanged and therefore are copied over to the active registers without modification.

Example #2 The host would like axis 1 to perform an emergency stop whenever the AxisIn signal for axis3 goes high. In addition the axis 1 acceleration should change whenever a particular commanded position is achieved on axis 4.

The following command sequence achieves this:

```
SetPosition Axis1, 123456             // Load destination
SetVelocity Axis1, 55555              // Load velocity
SetAcceleration Axis1, 500            // Load acceleration
SetDeceleration Axis1, 1000           // Load deceleration
Update Axis1                          // Make the move
SetBreakpointValue Axis1, 1, 0x400040 // Load mask and sense word of 0x40, 0x40 (bit 6 must
                                      // be high) for breakpoint 1
SetBreakpoint Axis1, 1, Axis3, AbruptStop, SignalStatus
                                      // Specify a breakpoint to monitor the signal status
                                      // register of axis 3 to trigger when bit 6 (AxisIn) goes
                                      // high for breakpoint 1
SetAcceleration Axis1, 111111         // Load a new acceleration of 111,111 but do not send
                                      // an Update
SetBreakpointValue Axis1, 2, 100000   // Load 100,000 into the comparison register
                                      // for breakpoint 2
SetBreakpoint Axis1, 2, Axis4, Update, PositiveCommandedPosition
                                      // Specify a positive commanded position breakpoint
                                      // on axis 4 which will result in an Update when satisfied
                                      // for  breakpoint 2
```

This sequence is similar to the previous one except that an additional breakpoint has been defined which causes the abrupt stop. In additional, these breakpoints have been setup to be triggered by events on axes 3 and 4. Both of these breakpoints were defined after the primary move was started although this may not be strictly necessary depending on when the breakpoint is expected to occur. Generally breakpoints should be set up after the primary move because there is only one set of buffered registers and it is thus impossible to load primary move parameters (position, velocity, etc.) and also "breakpoint" profile parameters (the profile parameters that will take effect once the breakpoint occurs) before the primary move is Updated.

6 Status Registers

6.1 Overview

The Navigator Motion Processor can monitor almost every aspect of the motion of an axis. There are numerous numerical registers that can be queried to determine the current state of the chipset, such as the current actual position (GetActualPosition command), the current commanded position (GetCommandedPosition command), etc.

In addition to these numerical registers there are three bit-oriented status registers which provide a continuous report on the state of a particular axis. These status registers conveniently combine a number of separate bit-oriented fields for the specified axis. These three 16-bit registers are Event Status, Activity Status, and Signal Status.

The host may query these three registers, or the contents of these registers may be used in breakpoint operations to define a triggering event such as "trigger when bit 8 in the signal status register goes low." These registers are also the source of data for the AxisOut (see Section 8.2) mechanism, which allows any bit within these three registers to be output as a hardware signal.

6.1.1 Event Status register

The Event Status register is designed to record events that do not continuously change in value, but rather tend to occur once upon some specific event. As such, each of the bits in this register is set by the chipset and cleared by the host.

The EventStatus register is defined in the table below:

Bit	Name	Description
0	Motion complete	Set when a trajectory profile completes. The motion being considered complete may be based on the commanded position or the actual encoder position. See section 6.3 for more details.
1	Position wraparound	Set when the actual motor position exceeds 7FFFFFFFh (the most positive position) and wraps to 80000000h (the most negative position), or vice versa.
2	Breakpoint 1	Set when breakpoint #1 is triggered.
3	Capture received	Set when the high-speed position capture hardware acquires a new position value.
4	Motion error	Set when the actual position differs from the commanded position by an amount more then the specified maximum position error.
5	Positive limit	Set when a positive limit switch event occurs.
6	Negative limit	Set when a negative limit switch event occurs.
7	Instruction error	Set when an instruction error occurs.
8-10	*Reserved*	*May contain 1 or 0.*
11	Commutation error	Set when a commutation error occurs (MC2300 series only).
12,13	*Reserved*	*May contain 1 or 0.*
14	Breakpoint 2	Set when breakpoint #2 is triggered.
15	*Reserved*	*May contain 0 or 1.*

The command GetEventStatus returns the contents of the Event Status register for the specified axis.

Bits in the Event Status register are latched. Once set, they remain set until cleared by a host instruction or a system reset. Event Status register bits may be reset to 0 by the instruction ResetEventStatus, using a 16-bit mask. Register bits corresponding to 0s in the mask are reset; all other bits are unaffected.

The event status register may also be used to generate a host interrupt signal using the SetInterruptMask command as described in Section 7.7.

6.1.2 Instruction error

Bit 7 of the event status register indicates an instruction error. Such an error occurs if an otherwise valid instruction or instruction sequence is sent when the Navigator's current operating state makes the instructions invalid. Instruction errors occur at the time of an update.

Should an instruction error occur the invalid parameters are ignored, and the Instruction Error indicator of the event status register is set. While invalid parameters checked at the time of the update are ignored, valid parameters are sent on. This can have unintended side effects depending on the nature of the motion sequence so all instruction error events should be treated very seriously.

Example: in the following sequence:

```
SetProfileMode (axis2, Velocity)
SetVelocity (axis2, -4387)
Update (axis2)
SetProfileMode (axis2, Trapezoidal)
SetPosition (axis2, 123456)
Update (axis2)
```

The negative velocity is not valid in the new profile mode. The Update is executed, but the Instruction Error bit is set. (Legitimate parameters, such as Position, are updated, and profile generation continues.)

6.1.3 Activity Status register

Like the Event Status register, the Activity Status register tracks various chipset fields.

Activity Status register bits however are not latched, they are continuously set and reset by the chipset to indicate the states of the corresponding conditions.

The ActivityStatus register is defined in the table below:

Bit	Name	Description
0	Phasing initialized	Set (1) when the motor's commutation hardware has been initialized. Cleared (0) if not yet initialized. Only valid for MC2300 series chipsets.
1	At maximum velocity	Set (1) when the commanded velocity is equal to the maximum velocity specified by the host. Cleared (0) if it is not. This bit functions only when the profile mode is trapezoidal, velocity contouring, or S-curve. It will not function when the chipset is in electronic gearing mode.
2	Position tracking	Set (1) when the servo is keeping the axis within the Tracking Window. Cleared (0) when it is not. See Section 6.2.
3-5	Current profile mode	These bits indicate the profile mode currently in effect, which might be different than the value set using the SetProfileMode command if an Update command has not yet been issued. The 3 bits define the current profile mode as follows: bit 5 bit 4 bit 3 Profile Mode 0 0 0 trapezoidal 0 0 1 velocity contouring 0 1 0 S-curve 0 1 1 electronic gear
6	Axis settled	Set (1) when the axis has remained within the Settle Window for a specified period of time. Cleared (0) if it has not. See Section 7.5.
7	*Reserved*	*May contain 0 or 1.*

Bit	Name	Description
8	Motor mode	Set (1) when the motor is "on", cleared (0) when the motor is "off." When the motor is on this means that the chipset can perform trajectory operations, and for the servo chipsets (MC2100, MC2300 and MC2800) it means the chipset is in closed loop mode and the servo loop is operating. When the motor is off this means trajectory operations cannot be performed and for the servo chipsets (MC2100, MC2300 and MC2800) it means the chipset is in open loop mode and the servo loop is disabled. The SetMotorMode command is normally used to select the mode of the motor, however the chipset will reset the mode to 0 (turn the motor off) if a motion error occurs.
9	Position capture	Set (1) when a new position value is available to read from the high speed capture hardware. Cleared (0) when a new value is not yet been captured. While this bit is set, no new values will be captured. The command GetCaptureValue retrieves a captured position value and clears this bit, allowing additional captures to occur.
10	In-motion indicator	Set (1) when the trajectory profile commanded position is changing. Cleared (0) when the commanded position is not changing. The value of this bit may or may not correspond to the value of the motion complete bit of the event status register depending on whether the motion complete mode has been set to commanded or actual.
11	In positive limit	Set (1) when the motor is in a positive limit condition. Cleared (0) when it is not.
12	In negative limit	Set (1) when the motor is in a negative limit condition. Cleared (0) when it is not.
13-15	S-curve segment	Indicates the S-curve segment number using values 1-7 to indicate S-curve phases 1-7 as shown in the S-curve trajectory section of this manual. A value of 0 in this field indicates the trajectory is not in motion. This field is undefined for profile modes other than S-curve, and may contain 0's or 1's.

The command GetActivityStatus returns the contents of the Activity Status register for the specified axis.

6.1.4 Signal Status

The signal status register provides real time signal levels for various chipset I/O pins. The SignalStatus register is defined in the table below:

Bit	Name	Description
0	A encoder	A signal of quadrature encoder input.
1	B encoder	B signal of quadrature encoder input.
2	Index encoder	Index signal of quadrature encoder input.
3	Home	Home position capture input.
4	Positive limit	Positive limit switch input.
5	Negative limit	Negative limit switch input.
6	AxisIn	Generic axis input signal.
7	Hall 1	Hall effect sensor input number 1 (MC2300 chipset only).
8	Hall 2	Hall effect sensor input number 2 (MC2300 chipset only).
9	Hall 3	Hall effect sensor input number 3 (MC2300 chipset only).
10	AxisOut	Programmable axis output signal.
11-15	*Reserved*	

The command GetSignalStatus returns the contents of the Signal Status register for the specified axis. All Signal Status register bits are inputs except bit 10 (AxisOut).

The bits in the signal status register always represent the actual hardware levels on the corresponding pins. A 1 in this register represents an electrically high value on the pin; a 0 indicates an electrically low level. The state of the signal sense mask affects the value read using GetSignalStatus (see the next section for more information on the signal sense mask).

6.1.5 Signal Sense Mask

The bits in the signal status register represent the high/low state of various signal pins on the chipset. How these signal pins are interpreted by the chipset may be controlled using a signal sense mask. This is useful for changing the interpretation of input signals to match the signal interpretation of the user's hardware.

The SignalSense mask register is defined in the table below:

Bit	Name	Interpretation
0	A encoder	Set (1) to invert quadrature A input signal. Clear (0) for no inversion.
1	B encoder	Set (1) to invert quadrature B input signal. Clear (0) for no inversion.
2	Index encoder	Set (1) to invert Index signal. Clear (0) for no inversion.
3	Home	Set (1) to invert home signal. Clear (0) for no inversion.
4	Positive limit	Set (1) for active high interpretation of positive limit switch, meaning positive limit occurs when signal is high. Clear (0) for active low.
5	Negative limit	Set (1) for active high interpretation of negative limit switch, meaning negative limit occurs when signal is high. Clear (0) for active low.
6	AxisIn	Set (1) to invert AxisIn signal. Clear (0) for no inversion.
7	Hall 1	Set (1) to invert Hall 1 signal. Clear (0) for no inversion.
8	Hall 2	Set (1) to invert Hall 2 signal. Clear (0) for no inversion.
9	Hall 3	Set (1) to invert Hall 3 signal. Clear (0) for no inversion.
10	AxisOut	Set (1) to invert AxisOut signal. Clear (0) for no inversion.
11-15	*Reserved*	

The command SetSignalSense sets the signal sense mask value. The command GetSignalSense retrieves the current signal sense mask.

7 Monitoring Motion Performance

7.1 Motion Error

Under certain circumstances, the actual axis position (encoder position) may differ from the commanded position (instantaneous output of the profile generator) by an excessive amount. Such an excessive position error often indicates a potentially dangerous condition such as motor or encoder failure, or excessive mechanical friction.

To detect this condition, thereby increasing safety and equipment longevity, the Navigator chipsets include a programmable maximum position error.

The maximum position error is set using the command SetPositionErrorLimit, and read back using the command GetPositionErrorLimit. To determine whether a motion error has occurred the position error limit is continuously compared against the actual position error. If the position error limit value is exceeded, then the axis is said to have had a "motion error."

At the moment a motion error occurs, several events occur simultaneously. The Motion Error bit of the event status word is set. If automatic stop on motion error is enabled the motor is set off, which has the effect of disabling the trajectory generator (for all chipsets). For the servo chipsets (MC2100, MC2300 and MC2800), it has the additional effect of disabling the servo loop and placing the chipset into open loop mode. This is the equivalent to a SetMotorMode *axis*, Off command.

To recover from a motion error that results in the motor being turned off, the cause of motion error should be determined and the problem corrected (this may require human intervention). The host should then issue a SetMotorMode *axis*, On command.

After the above sequence, the axis will be at rest, with the motor on.

If automatic stop on motion error is not set then only the motion error status bit is set, the motor is not stopped and no recovery sequence is required to continue operating the chipset. Nevertheless, for safety reasons, the user may still want to manually shut down the motion and explore the cause of the motion error.

7.1.1 Automatic Stop On Motion Error

Because a motion error may indicate a serious problem it is useful to have the axis automatically stop until the problem can be addressed and rectified. This feature is known as automatic stop on motion error.

The command SetAutoStopMode controls the action that will be taken if a motion error occurs. The options for this command are disable and enable.

If autostop is enabled, when a motion error occurs a SetMotorMode Off command is generated which has the effect of instantaneously halting the trajectory generator and (for servo chipsets) putting the chipset into open loop.

For stepping chipsets the motor will stop moving immediately (same as an abrupt stop). For servo chipsets (MC2100, MC2300 and MC2800) the trajectory will stop instantly but because motor off means open loop mode the motor will coast to a stop not under servo control in an amount of time determined by the velocity at the time of motion error and the inertia of the system.

For the servo chipsets, transitioning to open loop mode can be dangerous if the axis is oriented vertically, because the axis may fall downward due to gravity if not supported by the servo feedback. This problem can be rectified by use of the motor bias value, which is discussed in Section 4.1.2.

The motor bias is a fixed, open-loop command to the motor, which is added to the PID filter output. Upon a motion error with automatic stop enabled, the motor bias will be output even while the chipset is in open loop mode. Thus, with a properly set motor bias, if the axis experiences a motion error and transitions to open loop mode, the motor bias can prevent the axis from falling.

Caution: Because the motor bias value is applied 'open-loop' to the axis, care should be taken when setting its value.

7.2 Tracking window

The Navigator chipsets provide a programmable *tracking window* that can be used to monitor servo performance outside the context of a motion error. The tracking window functions similarly to the motion error, in that there is a programmable position error limit within which the axis must stay. Unlike the motion error facility however, if the axis moves outside of the tracking window the axis is not stopped. The tracking window is useful when external processes depend on the motor tracking the desired trajectory within some range. Alternatively the tracking window can be used as an early warning for performance problems that do not yet qualify as a motion error.

To set the size of the tracking window (maximum allowed position error to stay within the tracking window) the command SetTrackingWindow is used. The command GetTrackingWindow retrieves this same value.

When the position error is less than or equal to the window value the tracking bit in the activity status register is set. When the position error exceeds the tracking window value, the tracking bit is cleared. See Figure 7.5-1.

7.3 Motion Complete Indicator

In many cases it is useful to have the chipset signal that a given motion profile is complete. This function is available in the motion complete indicator.

The motion complete indicator appears in bit 0 of the event status register. As are all bits in the event status register, the motion complete bit is set by the chipset and cleared by the host. When a motion has been completed, the chipset sets the motion complete bit on. The host can examine this bit by polling the event status register or the host can program some automatic follow-on function using a breakpoint, a host interrupt, or an AxisOut signal. In either case, once the host has recognized that the motion has been completed the host should clear the motion complete bit, enabling the bit to indicate the end of motion for the next move.

Motion complete can indicate the end of the trajectory motion in one of two ways. The first is commanded; the motion complete indicator is set based on the profile generator registers only. The other method is actual, meaning the motion complete indicator is based on the actual encoder. The host instruction SetMotionCompleteMode determines which condition controls the indicator.

When set to commanded, the motion is considered complete when the trajectory generator registers for commanded velocity and acceleration both become zero. This normally happens at the end of a move when the destination position has been reached. But it may also happen as the result of a stop command (SetStopMode command), a change of velocity to zero, or when a limit switch event occurs.

When set to actual, the motion is considered complete when all of the following have occurred.

- The profile generator (commanded) motion is complete.

- The difference between the actual position and the commanded position is less than or equal to the value of the settle window. The settle window is set using the command SetSettleWindow. This same value may be read back using the command GetSettleWindow.
- The above two conditions have been met continuously for the last N cycles, where N is the programmed settle time. The settle time is set using the command SetSettleTime. This same value may be read back using the command GetSettleTime.

At the end of the trajectory profile the cycle timer for the actual-based motion complete mechanism is cleared, so there will always be at least an N cycle delay (where N is the settle time) between the profile generator being complete and the motion complete bit being set.

Appropriate software methods should be used with the actual motion complete mode because it is in fact possible that the motion complete bit will never be set if the servo is not tracking well enough to stay within the programmed position error window for the specified settle time.

The motion complete bit functions in the S-curve point-to-point, Trapezoidal point-to-point, and Velocity Contouring profile modes only. It does not function when the profile mode is set to Electronic Gearing.

7.4 In-motion indicator

The chipset can indicate whether or not the axis is moving. This function is available through the 'in-motion' indicator.

The in-motion indicator appears in bit 10 of the activity status register. The in-motion bit is similar to the motion complete bit however there are two important differences. The first is that (like all bits in the activity status register) the in-motion indicator continuously indicates its status without interaction with the host. In other words the in-motion bit cannot be set or cleared by the host. The other difference is that this bit always indicates the profile generator (commanded) state of motion, not the actual encoder.

The in-motion indicator bit functions in the S-curve point-to-point, Trapezoidal point-to-point, and Velocity Contouring profile modes only. It does not function when the profile mode is set to electronic gearing.

It is recommended that the Motion Complete bit be used for determining when a motion profile has finished.

7.5 Settled indicator

The chipset can also continuously indicate whether or not the axis has 'settled'.

The settled indicator appears in bit 6 of the activity status register. The settled indicator is similar to the motion complete bit when the motion complete mode is set to actual. The differences are that the settled indicator continuously indicates its status (cannot be set or cleared) and also that it indicates regardless of whether or not the motion complete mode is set to actual.

The axis is considered to be 'settled' when the axis is at rest (i.e. not performing a trajectory profile) and when the actual motor position has settled in at the commanded position for the programmed settle time.

The settle window and settle time used with the settled indicator are the same as for the motion complete bit when set to actual. Correspondingly the same commands are used to set and read these values: Set/GetSettleWindow, Set/GetSettleTime.

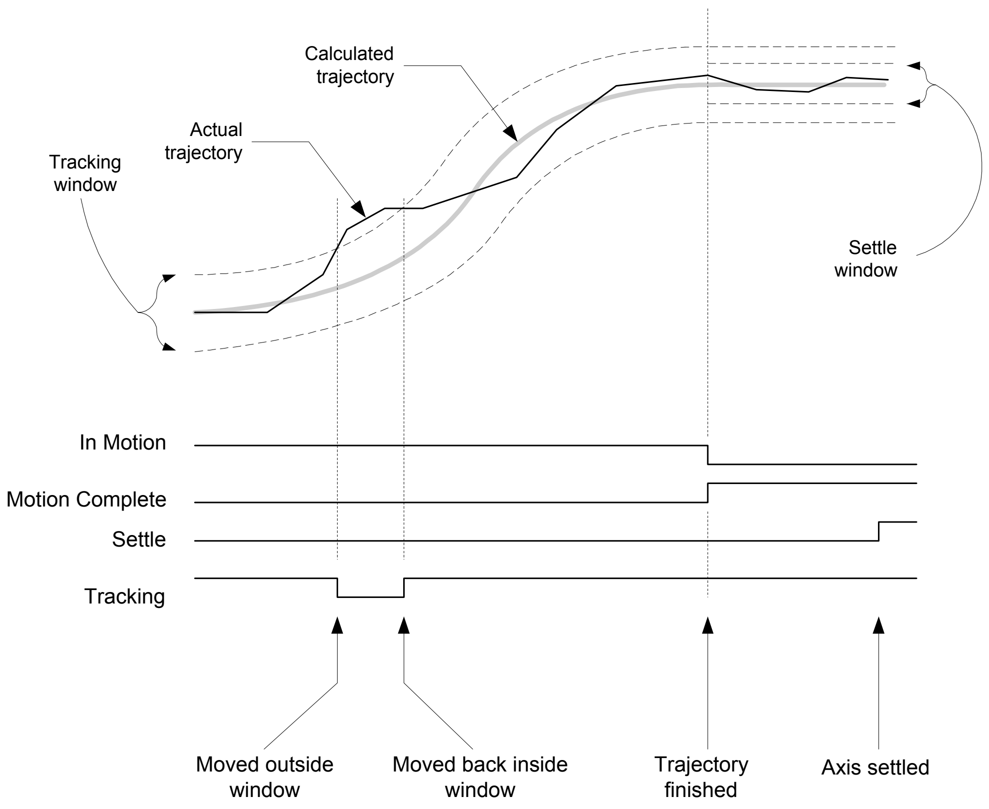


Figure 7.5-1. The tracking window

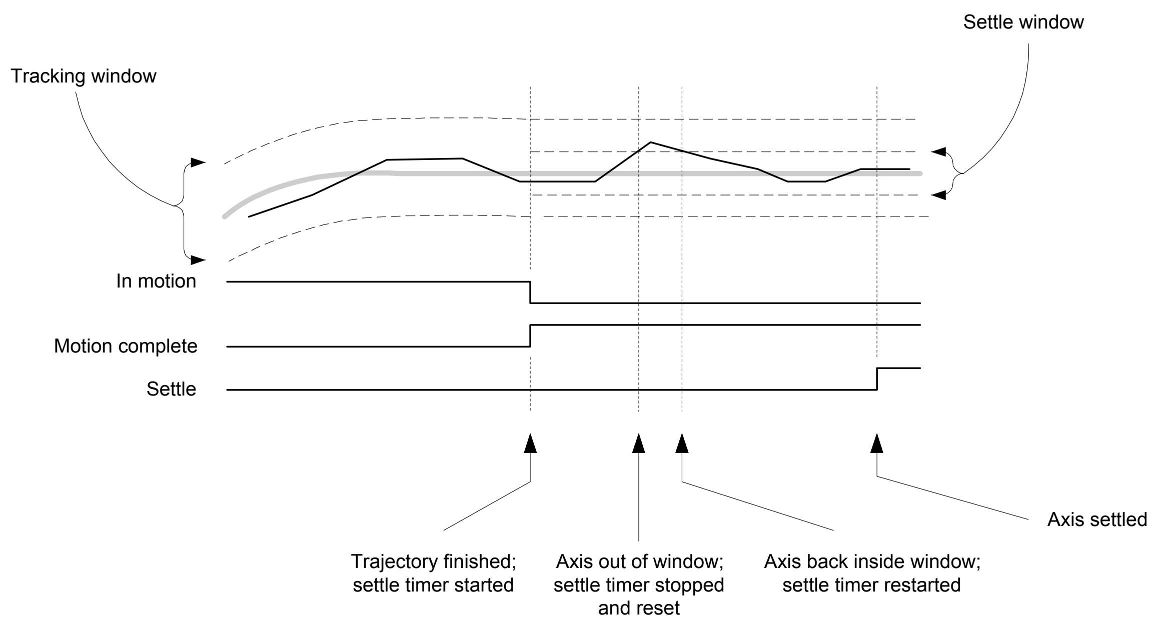

Settle window
Tracking window
In motion
Motion complete
Settle
Axis settled
Trajectory finished; settle timer started
Axis out of window; settle timer stopped and reset
Axis back inside window; settle timer restarted

Figure 7.5-2. The settle window

7.6 Data Trace

Data trace is a powerful feature of the Navigator chipset that allows various chipset parameters and registers to be continuously captured and stored to an external memory buffer. The captured data may later be downloaded by the host using standard memory buffer access commands. Data traces are useful for optimizing servo performance, verifying trajectory behavior, capturing sensor data, or to assist with any type of monitoring where a precise time-based record of the system's behavior is needed.

Generally, trace data capture (by the chipset) and trace data retrieval (by the host) are executed as two separate processes. The host specifies which parameters will be captured, and how the trace will be executed. Then the chipset performs the trace, and finally the host retrieves the data after the trace is complete. It is also possible however to perform continuous data retrieval even as the chipset is continuing to collect additional trace data.

To start a trace the host must specify a number of parameters. They are listed below:

Trace buffer - The host must initialize the data trace buffer memory. The Navigator chipset provides special instructions to initialize external memory into buffers, allowing various sizes and start locations of external memory to be used for tracing.

Trace variables - There are 27 separate items within the chipset that can be stored such as actual position, event status register, position error, etc. The user must select which variables, and from what axes, data will be recorded.

Trace period - The chipset can capture the value of the trace variables every single chipset cycle, every other cycle, or at any programmed frequency. The period of data collection and storage must be specified.

Trace mode - The chipset can trace in one of two modes; one-time, or rolling mode. This determines how the data is stored and whether the trace will stop automatically, or whether it must be stopped by the host.

Trace start/stop conditions - To allow precise synchronization of data collection it is possible to define the start and stop conditions for a given trace. The chipset monitors these specified conditions and starts or stops the trace automatically without host intervention.

7.6.1 The trace buffer

The Navigator chipset organizes external memory into data buffers. Each buffer is given a numerical ID. The trace buffer must always be ID 0 (zero). Before parameter traces may be used, memory buffer 0 must be programmed with a valid base address and length. Refer to 11.1.5 for more information.

The size of the trace buffer determines the maximum number of data points that may be captured. The maximum size of the trace buffer is only limited by the amount of physical memory in the system. The addressable memory space allows up to 2,048 Megawords of RAM to be installed, all of which (with the exception of the first 1K) may be used to store trace information.

While trace data is being collected, it is not legal to change the trace buffer configuration. If an attempt is made to change the base address, length, or write pointer associated with buffer 0 while a trace is running, the change will be ignored and an error will be flagged. It is, however, possible to change the read pointer and read data from the trace buffer while a trace is running. This allows the buffer to be constantly emptied while the trace runs.

7.6.2 The trace period

The tracing system supports a configurable *period register* that defines the frequency at which data is stored to the trace buffer. The tracing frequency is specified in units of chipset cycles, where one cycle is the time required to process all enabled axes.

The command SetTracePeriod sets the trace period, and the command GetTracePeriod retrieves it.

7.6.3 Trace variables

When traces are running, one to four chipset parameters may be stored to the trace buffer every trace period. The four *trace variable* registers are used to define which parameters are stored. Use the following commands to configure the trace variables:

The command SetTraceVariable selects which traceable parameter will be stored by the trace variable specified. The command GetTraceVariable retrieves this same value.

The value passed and returned by the preceding two commands specifies the axis and type of data to be stored. The format of this word is as follows:

Bits	Name	Description
0-1	Axis	Selects the source axis for the parameter.
2-7	Reserved	Must be written as zero.
8-15	ID	Selects the parameter to be stored.

The supported parameter ID values are:

ID	Name	Description
0	None	Indicates that no data is selected for the trace variable.
1	Position Error	The difference between the actual and commanded position values for the specified axis. This is the value used when evaluating the performance monitoring bits in the Activity Status register as well as the motion error bit in the Event Status register.
2	Commanded Position	The instantaneous commanded position output from the profile generator.
3	Commanded Velocity	The instantaneous commanded velocity output from the profile generator.
4	Commanded Acceleration	The instantaneous commanded acceleration output from the profile generator.
5	Actual Position	The actual position of the motor.
6	Actual Velocity	An estimated actual velocity (calculated using a simple low-pass filter).
7	Motor Command	Commanded motor output (output of the servo filter).
8	Chipset Time	The chipset time (units of servo cycles).
9	Capture Register	The current contents of the high speed capture register.
10	Servo Integral	The integral value used in the servo filter.
11	Servo Derivative	The derivative value used in the servo filter.
12	Event Status	The current contents of the event status register.
13	Activity Status	The current contents of the activity status register.
14	Signal Status	The current contents of the signal status register.
15	Phase Angle	The current motor phase angle (brushless motors only).
16	Phase Offset	The current phase offset value (brushless motors only).
17	Phase A Output	The value currently being output to motor winding 1.
18	Phase B Output	The value currently being output to motor winding 2. Only valid for two or three phase motors.
19	Phase C Output	The value currently being output to motor winding 3. Only valid for three phase motors.
20	Analog input 1	The most recently read value from analog input 1.
21	Analog input 2	The most recently read value from analog input 2.
22	Analog input 3	The most recently read value from analog input 3.
23	Analog input 4	The most recently read value from analog input 4.
24	Analog input 5	The most recently read value from analog input 5.
25	Analog input 6	The most recently read value from analog input 6.
26	Analog input 7	The most recently read value from analog input 7.
27	Analog input 8	The most recently read value from analog input 8.
28	PID Position Error	The difference between the actual and commanded position values for the specified axis. This is the value used as the error term for the PID filter calculation.

Setting a trace variable's parameter to zero will disable that variable **and all subsequent variables**. Therefore, if N parameters are to be saved each trace period, trace variables 0 – (N-1) must be used to identify what parameters are to be saved, and trace variable N must be set to zero. Note that $N \leq 4$.

For example; assume that the actual and commanded position values are to be stored for axis three each cycle period. The following commands would be used to configure the trace variables:

SetTraceVariable 0, 0203h	Sets trace variable 0 to store parameter 2 (commanded position) for axis 3.
SetTraceVariable 1, 0503h	Sets trace variable 1 to store parameter 5 (actual position) for axis 3.
SetTraceVariable 2, 0000h	Disables trace variables 2 (and above).

7.6.4 Trace modes

As trace data is collected it is written to sequential locations in the trace buffer. When the end of the buffer is reached, the trace mechanism will behave in one of two ways depending on the mode that has been selected.

If 'one-time' mode is selected then the trace mechanism will stop collecting data when the buffer is full.

If 'rolling-buffer' is selected then the trace mechanism will wrap around to the beginning of the trace buffer and continue storing data. In this mode the diagnostic trace will not end until the conditions specified in a SetTraceStop command are met.

Use the command SetTraceMode to select the trace mode. The command GetTraceMode retrieves the trace mode.

7.6.5 Trace start/stop conditions

The command SetTraceStart is used to specify what conditions will cause the trace mechanism to start collecting data. A similar command (SetTraceStop) is used to define the condition that will cause the trace mechanism to stop collecting data. Both SetTraceStart and SetTraceStop take a 16-bit word of data which contains four encoded parameters:

Bits	Name	Description
0-3	Trigger Axis	For trigger types other than *immediate*, this field determines which axis will be used as the source for the trigger. Use 0 for axis 1, 1 for axis 2, etc.
4-7	Trigger Type	Defines the type of trigger to be used. See the table below for a complete list of trigger types.
8-11	Bit Number	For trigger types based on a status register, this field determines which bit (0-15) of the status register will be monitored.
12-15	Bit State	For trigger types based on a status register, this field determines which state (0 or 1) of the specified bit will cause a trigger.

The Trigger Type field must contain one of the following values:

ID	Name	Description
0	Immediate	This trigger type indicates that the trace starts (stops) immediately when the SetTraceStart (SetTraceStop) command is issued. If this trigger type is specified, the trigger axis, bit number and bit state value are not used.
1	Update	The trace will start (stop) on the next update of the specified trigger axis. This trigger type does not use the bit number or bit state values.
2	Event Status	The specified bit in the event status register will be constantly monitored. When that bit enters the defined state (0 or 1) then the trace will start(stop).

3	Activity Status	The specified bit in the activity status register will be constantly monitored. When that bit enters the defined state (0 or 1) then the trace will start(stop).
4	Signal Status	The specified bit in the signal status register will be constantly monitored. When that bit enters the defined state (0 or 1) then the trace will start(stop).

7.6.6 Downloading Trace Data

Once a trace has been run and the trace buffer is full (or partially full) of data, this data may be downloaded by the host using the standard commands to read from external buffer memory. See section 10.1.5 for a complete description of external memory buffer commands.

When a trace stops running (either because of a SetTraceStop command or because the end of the trace buffer has been reached) the trace buffer's read pointer will be adjusted to point to the oldest word of data in the trace. If the trace buffer did not wrap then this will be location 0. If a wrap occurred in the trace buffer, then the read pointer will be set to the memory location that would have been overwritten by the next trace sample.

At any time, the command GetTraceCount can be used to get the number of 32-bit words of data stored in the trace buffer. This value may be used to determine the number of ReadBuffer commands that must be issued to download the entire contents of the trace buffer. Since the read pointer is automatically set to the oldest word of data in the trace buffer when the trace ends, and since the read pointer will automatically increment and wrap around the buffer as data is read, reading the entire contents of the trace buffer is as easy as issuing N ReadBuffer commands (where N is the value returned by GetTraceCount).

During each trace period, each of the trace variables is used in turn to store a 32-bit value to the trace buffer. Therefore, when data is read from the buffer, the first value read would be the value corresponding to trace variable 1, the second value will correspond to trace variable 2, up to the number of trace variables used.

Both the length of the trace buffer and the number of trace variables set directly affect the number of trace samples that may be stored. If for example the trace buffer is set to 1000 words (each 32-bits), and 2 trace variables are initialized (variables 0 and 1), then up to 500 trace samples will be stored. However, if three trace variables are used then 333 full trace samples may be stored. In this case the remaining word of data will store the first variable from the 334th sample.

If the trace mode is set to RollingBuffer then the 334th trace sample will store the first word in the last location of the trace buffer, and the second and third words will be stored in locations 0 and 1 respectively. In this case the first two words of the first sample have been overwritten by the last two words of sample 334. When the trace is stopped the read pointer will point to the oldest word of data in the buffer. This word may not correspond to the first word of a trace sample.

It is therefore recommended that the length of the trace buffer be set so that it is an even multiple of the number of trace variables being used. This will ensure that the read index is pointing to the first word in a complete trace sample whether the trace buffer wraps or not. The simplest solution is to make sure that the trace buffer length is an even multiple of 12 (since 12 is evenly divisible by all possible numbers of trace variables 1, 2, 3 or 4).

7.6.7 Running Traces

Here is a summary of data trace operation to help you get started:

1. Specify what data is to be stored. The command SetTraceVariable is used to specify up to four variables to be stored each trace period. Trace variables locations must be used contiguously. For example, to trace two variables use trace variables 0 and 1. To trace three variables use trace

variables 0, 1 and 2. The first trace variable found that is set to *none* (refer to **SetTraceVariable** in the *Programmer's Reference*) is assumed to be the last variable being traced. If trace variable 0 is set to *none* and trace variable 1 is set to *actual position*, no variables will be traced since the first variable (set to *none*) specifies the end of variables being traced. If four variables are being traced, do not set any variables to *none* since all variable locations are being used.

2. Setup the trace buffer. Using the commands **SetBufferStart** and **SetBufferLength**, define the location in external RAM where trace data should be stored and the amount of RAM to be used to hold the trace data. The trace data will be stored in the buffer with an ID of zero. Be careful not to extend the buffer beyond the amount of physical RAM available in your system, in particular recall that **SetBufferStart** and **SetBufferLength** specify values based on a 32-bit word size.
3. Set the trace period. The command **SetTracePeriod** sets the interval between trace samples in units of chip cycles. The minimum is one cycle.
4. Set the trace mode. If the trace is to be started on some event then the OneTime mode should probably be used. This will allow one buffer full of trace data to be stored beginning with the starting event (set using **SetTraceStart**). Alternatively, if the trace is to stop on an event (as specified using **SetTraceStop**) then the rolling buffer mode should be used. This will cause the system to constantly record data until the stopping event occurs. At that point the data leading up to the event will be saved in the trace buffer.
5. Set the stopping mode (if desired). If a specific event will cause the trace to stop, then it should be programmed using the **SetTraceStop** command. However, if the trace is to be programmed to stop when the buffer fills up (by setting the trace mode to OneTime) then it is not necessary to set another stopping event. Also, at any time while the trace is running the **SetTraceStop** command may be issued to stop the trace immediately.
6. Start the trace. The **SetTraceStart** command may be used to start the trace directly (by specifying the immediate trigger type). Alternatively, a triggering event may be specified which will start the trace when it occurs.

7.7 Host Interrupts

Interrupts allow the host to become aware of a special chipset condition without the need for continuous monitoring or polling of the status registers. The Navigator chipsets provide this service in the form of a *host interrupt*.

The events that trigger a host interrupt are the same as those that set the assigned bits of the Event Status Register—reproduced here for convenience.

Bit	Event	Occurs when
0	Motion complete	The profile reaches its endpoint, or motion is otherwise stopped.
1	Position wraparound	The axis position wraps.
2	Breakpoint 1	Breakpoint 1 condition has been satisfied.
3	Capture received	Encoder index pulse or home pulse has been captured.
4	Motion error	Maximum position error set for a particular axis has been exceeded.
5	Positive limit	Positive over travel limit switch violation has occurred.
6	Negative limit	Negative over travel limit switch violation has occurred.
7	Instruction error	Host instruction causes an error.
11	Commutation error	*[MC2300 only]* Index pulse does not match actual phase.
14	Breakpoint 2	Breakpoint 2 condition has been satisfied.

Using a 16-bit mask, the host may condition any or all of these bits to cause an interrupt. This mask is set using the command SetInterruptMask. The value of the mask may be retrieved using the command GetInterruptMask. The mask bit positions correspond to the bit positions of the event status register. If a 1 is stored in the mask then a 1 in the corresponding bit of the event status register will cause an interrupt to occur. Each axis supports its own interrupt mask, allowing the interrupting conditions to be different for each axis.

The chipset continually scans the event register and interrupt mask together to determine if an interrupt has occurred. When an interrupt occurs, the HostIntrpt signal is made active.

At this point the host can respond to the interrupt (although the execution of the current host instruction, including the transfer of all associated data packets, should be completed), but it is not required to do so.

Since it is possible for more then one axis to be configured to generate interrupts at the same time, the chipset provides the command GetInterruptAxis. This command returns a bitmasked value with one bit set for each axis currently generating an interrupt. Bit 0 will be set if axis 1 is interrupting, bit 1 is set for axis 2, etc. If no interrupt is currently pending then no bits will be set.

To process the interrupt, normal chipset commands are used. The specific commands sent by the host to process the interrupt depend on the nature of the interrupting condition, however at a minimum the interrupting bit in the event status register should be cleared using the ResetEventStatus command. If this is not done then the same interrupt will immediately occur once interrupts are re-enabled.

Once the host has completed processing the interrupt, it should send a ClearInterrupt command to clear the interrupt line, and re-enable interrupt processing. Note that if another interrupt is pending the interrupt line will only be cleared momentarily and then reasserted.

Following is a typical sequence of interrupts and host responses. In this example, an axis has hit a limit switch in the positive direction causing a limit switch event and an abrupt stop. The abrupt stop causes a motion error. Assume that these events all occur more or less simultaneously. In this example the interrupt mask for this axis has been set so that either motion errors or limit switch trips will cause an interrupt.

Event	Host action
Motion error & limit switch trip generates interrupt	Sends GetInterruptAxis instruction.
A bitmask identifying all interrupting axes is returned by the chipset. This value identifies one axis as generating the interrupt.	Sends GetEventStatus instruction, detects motion error and limit switch flags are set. Issues a ResetEventStatus command to clear both bits. Returns the axis to closed loop mode by issuing SetMotorMode On command. Issues a ClearInterrupt command to reset the interrupt signal.
Chipset clears motion error bit and disables host interrupt line	Generates a negative direction move to clear the limit switch.
Motor moves off limit switch. Activity status limit bit is cleared.	None

At the end of this sequence, all status bits are clear, the interrupt line is inactive, and no interrupts are pending.

8 Hardware Signals

There are a number of signals that appear on each axis of the Navigator chipsets, which can be used to coordinate chipset activity with events outside the chipset. In this section we will discuss these signals. They are the bidirectional travel limit switches, the AxisIn pin, and the AxisOut pin. These signals appear on each axis of the chipset. For example, a four-axis chipset such as the MC2140 has four AxisIn pins, four positive limit switches, etc.

8.1 Travel-limit switches

The Navigator chipsets support motion travel limit switches that can be used to automatically recognize an "end of travel" condition. This is an important safety feature for systems that have a defined range of motion.

The following figure shows a schematic representation of an axis with travel-limit switches installed, indicating the "legal" motion area and the over-travel or illegal region.

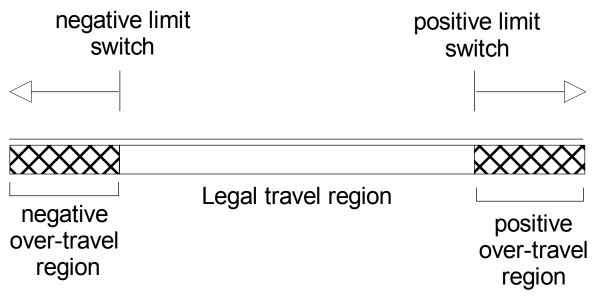


Figure 8.1-1 Directional limit switch operation

The positive and negative switches are connected to CP chip inputs PosLim1-4 and NegLim1-4, to detect over-travel in the positive and negative directions, respectively.

There are two primary functions that the Navigator provides in connection with the over-travel limit switch inputs. The host can be automatically notified that an axis has entered an over-travel condition, allowing the host to take appropriate special action to manage the over-travel condition.

Upon entering an over-travel condition, the trajectory generator can be automatically halted, so that the motor does not travel further into the over travel region.

Limit switch processing may be enabled or disabled for a given axis through the command SetLimitSwitchMode. This same register can be read using the command GetLimitSwitchMode.

If limit processing is enabled then the chipset will constantly monitor the limit switch input pins looking for a limit switch event. A limit switch event occurs when a limit switch goes active while the axis commanded position is moving in that limit switch's direction. If the axis is not moving, is in open-loop mode, or is moving in the opposite direction, then a limit switch event will not occur. For example a positive limit switch will occur when the axis commanded position is moving in the positive direction and the positive limit switch goes active. However it will not occur if the axis commanded position is moving in the negative direction or is stationary.

The "sense" of the limit switch inputs (active high or active low) can be controlled using the SetSignalSense command.

When a limit event occurs, the chipset will generate an abrupt stop thereby halting the motion. To prevent any accidental motion, the trajectory velocity is set to zero, equivalent to a SetVelocity 0 command. In addition, the bit in the event status register corresponding to the active limit switch will be set. Finally, the appropriate bit in the activity status register will be set.

Once an axis has entered a limit switch condition the following steps should be taken to clear the limit switch event:

- Unless limit switch events can occur during normal machine operation the cause of the event should be investigated and appropriate safety corrections made.
- The limit switch bit(s) in the event status register should be cleared by issuing the ResetEventStatus command. No motion is possible in any direction while either of the limit switch bits in the event status register is set.
- A move should be made in the direction opposite to the direction that caused the limit switch event. This can be any profile move that 'backs' the axis out of the limit. If the host instead attempts to move the axis further into the limit, a new limit event will occur and an instruction error will be generated. (See section 6.1.2 on instruction errors for more information).

If the limit switches are wired to separate switches it should not be possible for both limit switches to be active at the same time. However, if this does occur (presumably due to a special wiring arrangement) then both limit switch bits in the activity status register would be set, thus disabling moves in either direction. In this case, the SetLimitSwitchMode command should be used to temporarily disable limit switch processing while the motor is moved off of the switches.

NOTE: Limit switches do not function when the chipset is in 'motor off', also known as 'open-loop' mode.

8.2 The *AxisOut* pin

Each axis has a general purpose axis output pin which can be programmed to track the state of any of the assigned bits in the Event Status, Activity Status, or Signal Status registers. The tracked bit in one of these three registers may be in the same axis or in a different axis as the axis of the AxisOut pin itself. This function is useful for outputting hardware signals to trigger external peripherals.

The chipset command SetAxisOutSource may be used to configure the axis output pin. This command takes a single word as an argument. The value of this parameter is interpreted as follows:

Bits	Name	Description
0-3	Source axis	Specifies the axis to be used as a source for the axis output signal. The axis output pin will follow the specified register bit of the source axis. Writing a zero indicates axis 1, writing a one indicates axis 2, etc.
4-7	Bit number	Indicates which bit in the selected status register will be followed by the axis output pin. Bits are numbered from 0 to 15 where bit 0 indicates the least significant bit.
8-11	Status register	Indicates which register will be used as the source for the axis output. The encoding is as follows: **ID** **Register** 0 None, the axis output pin will always be inactive 1 Event status register 2 Activity status register 3 Signal status register 4-15 Reserved, do not use
12-15	Reserved	Reserved for future use. Should be written as zeros.

Note that the AxisOut pin may be configured to be active low or active high using the SetSignalSense command.

It is possible to use the AxisOut pin as a software- programmed direct output bit under direct host control. This can be done by selecting zero as the register ID code in the SetAxisOutSource command and by adjusting the level of the resulting inactive output state to high or low as desired using the SetSignalSense command.

8.3 The *AxisIn* pin

Each axis has an input pin (AxisIn) which can be used as a general purpose input, readable using the GetSignalStatus command, as well as to trigger automatic events such as performing a motion change (stop, start, change of velocity, etc.) upon a signal transition using breakpoints.

No special commands are required to setup up or enable the AxisIn signals.

8.4 Analog input

The Navigator motion processors provides 8 channels of general-purpose analog input. These 8 inputs are connected to internal circuitry that converts the analog signal to a digital word with a precision of 10-bits. The conversions are performed continuously, with all 8 channels being converted every 4 cycles of the chip. For the MC2100/MC2500 this is every 4x100us, and for the MC2300/MC2400/MC2800 this is every 4x150us.

To read the most recent value converted by any of the 8 channels, the command ReadAnalog is used. The value returned by this command is the result of shifting the converted 10-bit value 6 bits left.

The analog data is general-purpose and is not used by the motion processor in any calculations.

For information on electrical characteristics, please refer to the *Navigator Technical Specifications Manual* appropriate for your motion processor.

8.5 The *Synch* pin - multiple chip synchronization (MC2xx3 only)

The Navigator MC21x3, MC23x3, MC24x3 and MC28x3 motion processors contain support for synchronizing their internal cycle time. This allows the start/stop and modification of motion profiles to be synchronized across chipsets where precise timing of movement is required. This may be necessary because the chipsets are remotely located or the application may require greater than 4 axes of synchronized motion. In the most common configuration one chipset is assigned as a Master node and other chipsets are set into Slave mode.

The input/output state of this pin and its function are set using the command SetSynchronizationMode *mode*

The table below summarizes the modes.

Encoding	Mode	Description
0	Disabled	In the disabled mode, the pin is configured as an input and is not used. This is the default mode after reset or power up.
1	Master	In the master mode, the pin outputs a synchronization pulse that can be used by slave nodes or other devices to synchronize with the internal chip cycle of the master node.
2	Slave	In the slave mode, the pin is configured as an input and a pulse on the pin synchronizes the internal chip cycle.

When synchronizing multiple chipsets, the following rules must be observed:-

- All chipsets must have their sample time set to the same value. For example, if an MC2143 and MC2123 are to be synchronized the sample time must be at least the greater of the two sample times. In this case the value must be at least 400us since the MC2143 has a sample time of 400us and the MC2123 has a sample time of 200us.
- Only one node in the network can be a master. For example, with 3 chipsets in a network one chipset must be designated the master and the other two must be slaves.
- Slave nodes must be set into slave mode before the master is set as the master node. This ensures that the slave(s) cycle starts at precisely the moment that the master assumes its state as master.

Since the Pilot Motion Processor also contains the synch feature, Navigator and Pilot processors can be mixed together in a design and synchronized as long as the rules shown above are followed.

The following sequence of C-Motion commands demonstrates the required setup sequence for enabling 2 chipsets to operate in synch mode.

```
// make sure board#1 (master node) is not outputting a synch signal yet
PMDSetSynchronizationMode( &hBoard1_Axis1, PMD_SynchDisabled );

// make sure the slave board has the same sample time as the master
PMDGetSampleTime( &hBoard1_Axis1, &sample_time );
PMDSetSampleTime( &hBoard2_Axis1, sample_time );

// set board#2 into slave mode
PMDSetSynchronizationMode( &hBoard2_Axis1, PMD_SynchSlaveNode );

// board#2's internal timer is now stalled and set to zero

// set board#2 into master mode
PMDSetSynchronizationMode( &hBoard1_Axis1, PMD_SynchMasterNode );

// the two chipsets are now synchronized
```

Once the above sequence of instructions has been executed, the internal time on the master and slave(s) will be the same. This time breakpoint mechanism is then used to execute changes of motion on all chipsets in the network at precisely the same time.

The following sequence of C-Motion commands demonstrates the required setup sequence for starting a synchronized move across two chipsets.

```
// buffer a new destination position for board#1, axis#1
PMDSetPosition( &hBoard1_Axis1, 50000 );

// buffer a new destination position for board#2, axis#1
PMDSetPosition( &hBoard2_Axis1, 250000 );

// Use the master nodes time to set breakpoints for both boards
// that will start motion 20 chip cycles from now
PMDGetTime( &hBoard1_Axis1, &time );
PMDSetBreakpointValue( &hBoard1_Axis1, time+20 );
PMDSetBreakpoint( &hBoard1_Axis1, PMDBreakpoint1, PMDAxis1,
                    PMDBreakpointActionUpdate, PMDBreakpointTime );
PMDSetBreakpointValue( &hBoard2_Axis1, time+20 );
PMDSetBreakpoint( &hBoard2_Axis1, PMDBreakpoint1, PMDAxis1,
                    PMDBreakpointActionUpdate, PMDBreakpointTime );

// when the breakpoint triggers (20 chip cycles from now)
// both axes will start motion
```

The master node outputs a synchronization pulse once every 50us. Applications are free to use this pulse to synchronize external equipment to the internal cycle of the PMD chipset.

9 Motor Interfacing

The Navigator chipsets support two types of motor position input: incremental-encoder and parallel-word feedback. The feedback type is programmable, but the appropriate hardware must be installed for the selection to be effective.

To set the feedback type the command SetEncoderSource is used. This same value can be read back using the command GetEncoderSource.

9.1 Incremental Encoder Input

Incremental encoder feedback provides two square-wave signals: A quadrature, B quadrature, and an optional Index pulse that normally indicates when the motor has made one full rotation. The A and B signals are offset from each other by 90°, as shown in Figure 9.1-1.

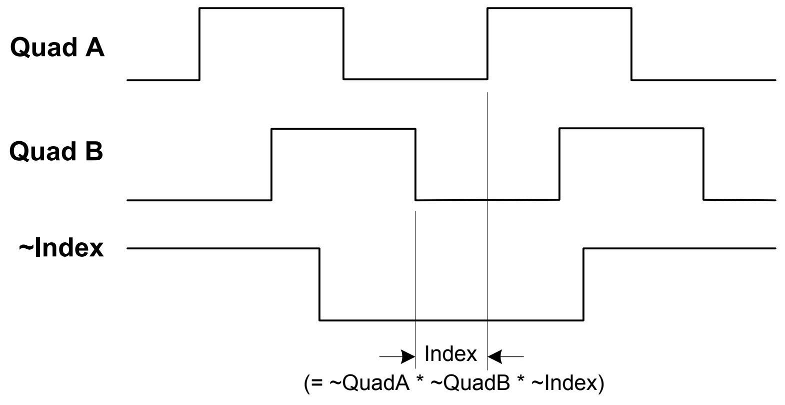


Figure 9.1-1. Quadrature Encoder Timing

For the quadrature incremental position to be properly registered by the chipset, when the motor moves in the positive direction QuadA should lead QuadB. When the motor moves in the negative direction, QuadB should lead QuadA. Because of the 90° offset, four resolved quadrature counts occur for one full phase of each A and B channel.

9.1.1 Actual Position Register

The chipset continually monitors the position feedback signals and accumulates a 32-bit position value called the Actual Position. Upon powerup the default Actual Position is zero. The Actual Position can be explicitly set using the command SetActualPosition, and can be retrieved using the command GetActualPosition.

Particularly when using incremental feedback, the Actual Position is generally set shortly after powerup, using a homing procedure to reference the actual position to a physical hardware location.

9.1.2 Digital Filtering

All encoder inputs as well as both the index and home capture inputs, are digitally filtered to enhance reliability. The filter requires that a valid transition be accepted only if it remains in its new (high or low) state for at least three cycles of 50 nsec each (total 150 nsec). This insures that brief noise pulses will not be interpreted as an encoder transition.

Although this digital filtering scheme can increase the overall reliability of the quadrature data, to achieve the highest possible reliability additional techniques may be required, such as differential line drivers/receivers, or analog filtering. Whether these additional schemes are required depends on the specific system, and the amount and type of noise sources.

9.1.3 High speed position capture

Each axis of the Navigator chipsets supports a high-speed position capture register that allows the current axis location to be captured, triggered by an external signal. One of two signals may be used as the capture trigger: the index signal or the home signal.

These two input triggers differ in how a capture is recognized. If the index signal is used as the trigger, a capture will be triggered when the A, B, and Index signals achieve a particular state (defined by the signal sense register using the SetSignalSense command). If the home signal is selected as the capture trigger source then only the home signal need be in a particular state for the capture to be triggered.

The default values for the A, B, index, and home sense signals in the signal sense register are 0, meaning these signals are active low. In this condition if index is selected as the trigger source a capture will be recognized when A, B, and Index are all low. Any change to the sense mask from active low interpretation to active high interpretation will cause a corresponding change in recognition of the trigger condition.

The command SetCaptureSource determines whether the index signal or the home signal will be used as the position capture trigger. The command GetCaptureSource retrieves this same value.

When a capture is triggered, the contents of the actual position registers are transferred to the position capture register, and the capture-received indicator (Bit 3 of the Event Status register) is set. To read the capture register the command GetCaptureValue is used. The capture register must be read before another capture can take place. Reading the position capture register causes the trigger to be 're-armed', meaning more captures can thereafter occur. As for all event status register bits, to clear the position capture indicator the command ResetEventStatus is used.

9.2 Parallel-word position input

For feedback systems that do not provide incremental signals, but instead a digital binary word, the Navigator supports a parallel-word input mechanism which can be used with a large variety of devices including Resolvers (after Resolver to Digital conversion), absolute optical encoders, laser interferometers with parallel word read-out, incremental encoders with external quadrature decoder circuit and A/D converters reading an analog feedback signal.

In this position-input mode the encoder position is read through the chipset's external bus by reading a 16-bit word, one for each axis set to this mode. Depending on the nature of the feedback device fewer than 16 bits of resolution may be available, in which case the unused high order data bits should be arranged to indicate a 0 value when read by the chipset. It is also acceptable to sign extend these bits, however under no circumstances should unused bits of the parallel-word be left floating.

The value input by the chipset should be binary coded. The Navigator chipsets assume that the position data provided by the external device is a two's complemented signed number although if the value returned instead ranges from 0 to 2^n-1 (where n is the number of bits provided by the feedback device) then the difference in behavior will be the interpretation of the start location, which will be 'shifted' by 1/2 the full scale feedback range. If desired this initial position may be altered using the SetActualPosition command.

9.2.1 Multi-turn systems

In addition to supporting position tracking across the full numeric feedback range of a particular device the ability to support multi-turn systems is also provided. The parallel encoder values being read in are continuously examined and a position "wrap" condition is automatically recognized, whether from largest encoder value to smallest encoder value (negative wrap) or from smallest value to largest value (positive wrap).

Using this "virtual" multi-turn counter, the Navigator chipsets continuously maintain the axis location to a full 32 bits. Of course if the axis does not wrap around (non multi-turn system), the range will stay within a 16 bit value.

As the motor moves in the positive direction, the input value should increase up to some maximum at which point it may wrap back to zero and continue increasing from there. Likewise, when the motor moves in the negative direction the value should decrease to zero at which point it may wrap back to its maximum. The value at which the parallel input device wraps is called the device's modulus, and should be set using the SetEncoderModulus command. Note that the SetEncoderModulus command takes as a parameter ½ the value of the modulus.

For example if a rotary motor uses a 12-bit resolver for feedback the encoder modulus is 4,096, and therefore the value sent to the SetEncoderModulus command would be 2,048. Once this is done each time the motor rotates and the binary word value 'jumps' from the largest binary output to the smallest, the Navigator chipset will properly recognize the motor wrap condition and accumulate actual encoder position with values larger than 4,096 or smaller than 0.

For systems that use a position counter with a modulo smaller than the encoder counts per revolution, set the counts/rev value equal to the position counter size. For example, if a rotary laser interferometer is being used which provides a 16-bit output value, but provides 16,777,216 counts per revolution, use a counts/rev value of 32,768 ($2^{16}/_{2}$).

No high-speed position capture is supported in the parallel-word device input mode. Therefore the index and home signals, as well as the quadrature A and B signals, are unused.

Nevertheless for other non-position capture functions utilizing the signal status register these signals can still be used in the normal way. For example, these bits can be read (GetSignalStatus command) or these bits can be used as breakpoint triggers, etc.

9.2.2 Parallel-word device interfacing

For each axis that is set for parallel-word input the chipset will use its peripheral bus and the addresses listed below to read the position feedback value for that axis. The value is read from the chipset's peripheral address bus at the following addresses:

Peripheral bus address	Chipset I/O operation	Addressed motor (Axis #)
800h	16-bit peripheral read	1
801h	16-bit peripheral read	2
802h	16-bit peripheral read	3
803h	16-bit peripheral read	4

To perform the read, the chipset will drive the peripheral bus signals as detailed in Section 9.3.

For each axis that is set to parallel-word position input mode the chipset will perform a peripheral read at the corresponding address, but axes not in parallel-word mode will not be addressed. This read occurs every 50µs. For example, if Axes 1 and 3 are set to incremental feedback mode and 2 and 4 are set to parallel-word, then Addresses 0801h (address for axis #2 from above table) and address 0803h (address for axis #4) will be read by the chipset every 50µs, and the values returned by the associated host peripheral circuitry will determine the actual position for axis 2 and 4.

9.3 Peripheral Device I/O

The Navigator chipsets make use of an external peripheral bus to write to and read from the user's peripheral devices associated with various dedicated chipset functions such as DAC (Digital to Analog Converter) signal output, parallel-word input, user-defined memory I/O, and serial port configuration input.

The bus consists of 5 control signals; ~PeriphSlct, ~Strobe, W/~R, R/~W, and ~WriteEnbl. In addition there are 16 address lines (Addr0-Addr15) and 16 data lines (Data0-Data15). The chipset manipulates these control, address, and data lines in such a way that the user can develop peripheral circuitry to read or write data into or from the chipset.

9.3.1 Peripheral Device Read

To perform a peripheral read the chipset will drive to a low level the ~PeriphSlct signal, the ~Strobe signal, and the W/~R signal. In addition the chipset will drive to a high level one or more address bits depending on the number of addresses to decode for a given function. After these signals have been asserted it is expected that the user's peripheral circuitry will assert the correct data onto the data bus Data0-15.

The following table summarizes the addresses of the Navigator peripherals:

Base Address	Address bits to decode	Device	Description
0200h	A9	Serial port data	Contains the configuration data (transmission rate, parity, stop bits, etc) for the asynchronous serial port.
0800h - 0803h	A0, A1, A11	Parallel-word encoder	Addresses for parallel-word feedback devices for axes 1-4.
1000h - 10ffh	A0-A7, A12	User-defined	Addresses for user-defined I/O devices.
2000h	A13	RAM page pointer	Page pointer to external memory.
4000h - 4007h	A0 - A2, A14	Motor-output DACs	Addresses for motor-output D/A converters for Axes 1 - 4.

The specific timing for the Peripheral Read function is detailed in the Navigator Technical Specification manual.

9.3.2 Peripheral Device Write

To perform a peripheral write the chipset will drive to a low level the ~PeriphSlct signal, the R/~W signal, and the ~WriteEnbl signal. In addition the chipset will drive to a high level one or more address bits depending on the number of addresses to decode for a given function, also asserting the data word to be written by the chipset.

After the Navigator asserts these signals it is expected that the user's peripheral circuitry will read in the data written by the chipset.

The specific timing for the Peripheral Write function is detailed in the Navigator Technical Specification manual.

9.4 Motor Command Output

The Navigator chipsets provide a variety of methods to interface to the motor amplifier, however the methods available vary with each Navigator chipset. The following table shows the motor output options available for each Navigator chipset:

Series	Motors supported	Number of phases per axis	Available output formats
MC2100	Brushed Brushless servo (with external commutation)	1	Sign/Magnitude PWM 50/50 PWM 16-bit DAC
MC2300	Brushless servo	2 or 3	Sign/Magnitude PWM* 50/50 PWM 16-bit DAC
MC2400	Microstepping motor	2 or 3	Sign/Magnitude PWM* 50/50 PWM 16-bit DAC
MC2500	Stepping motor	1	Pulse and direction.
MC2800	Brushed servo and brushless servo motors	1 for brushed servo motor 2 or 3 for brushless servo motor	Sign/Magnitude PWM* 50/50 PWM 16-bit DAC

** only when 2-phases are selected. This format is not supported for 3-phase motors.*

Sign magnitude PWM, 50/50 PWM, and 16-bit DAC all output a signed numerical motor command value although using different signal formats to accomplish this. Each of these three signal formats 'encodes' this signed numerical value which represents the torque or velocity at which the chipset is commanding the motor.

Pulse and direction is fundamentally different from the other three representations because it makes no attempt to encode a motor torque. Instead pulse and direction supports step motor amplifiers which accept a positive direction movement "pulse" and a negative direction movement "pulse." The amplifier then determines the motor torque, usually through an analog setting on the amplifier itself.

The command SetOuputMode controls which of the available output formats will be used. The command GetOutputMode retrieves this same value.

9.4.1 Sign magnitude PWM

In this mode, two pins are used to output the motor command information for each motor phase. One pin carries the PWM magnitude that ranges from 0 to 100%. This signal expresses the absolute magnitude of the desired motor command. A high signal on this pin means the motor coil should be driven with voltage. A second pin outputs the sign of the motor command by going high for positive sign and low for negative.

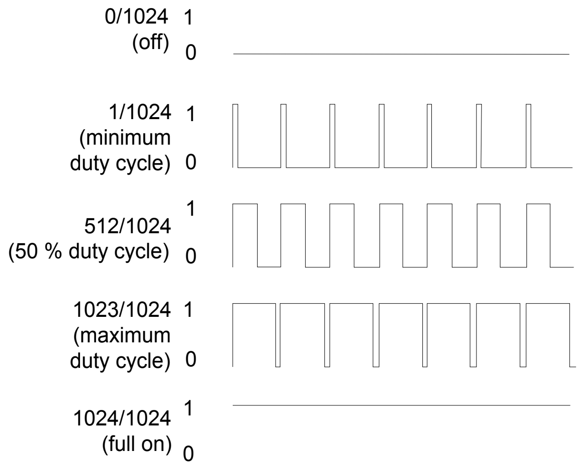
In this mode, output is resolvable to 1 part per 2,048. The total range of the motor command register which is -32,768 to +32,767 is scaled to fit the PWM output range.

For example if the motor command that the chipset outputs for a given phase is +12,345 then the sign bit will be output as a high level, and the magnitude pin will output with a duty cycle of 2,048*12,345/32,768 = 771.56 = 772, meaning the magnitude signal will be high for 772 cycles and low for the remaining 1,276 cycles. If it were desired that the output value be -12,345 then the magnitude signal would be the same but the sign bit would be low instead of high.

Sign magnitude PWM output is typically used with H-bridge type amplifiers. Most such amplifiers have separate sign and magnitude inputs allowing the Navigator signals to be connected directly.

The following figure shows sign and magnitude output wave forms:

Figure 9.4-1 shows some typical PWM output waveforms.



0/1024 (off)
1
0
1/1024 (minimum duty cycle)
1
0
512/1024 (50 % duty cycle)
1
0
1023/1024 (maximum duty cycle)
1
0
1024/1024 (full on)
1
0

Figure 9.4-1. Typical Sign/Magnitude PWM output signals

9.4.2 50/50 PWM

In this mode only one pin is used per motor output or per motor phase. This pin carries a variable duty cycle PWM signal much like the magnitude signal for sign magnitude PWM. This PWM output method differs however in that a 50% output signal (high half the time, low half the time) indicates a desired motor command of zero, and positive motor commands are encoded as duty cycles greater than 50% duty cycles, and negative motor commands are encoded as duty cycles less than 50 %. In this mode a full on positive command is encoded as a 100% duty cycle (always high) and a full on negative command is encoded as a 0 % duty cycle (always low).

For example if the motor command that the chipset outputs for a given phase is +12,345 then the magnitude pin will output with a duty cycle of 1,024 + 1,024*12,345/32,768 = 771.56 = 1,409.78 = 1,410, meaning the magnitude signal will be high for 1,410 cycles and low for the remaining 638 cycles. If it were desired that the output value be -12,345 then the magnitude signal would have a duty cycle of 1,024 + 1,024*-12,345/32,768 = 638.2 = 638 meaning the magnitude signal will be high for 638 cycles and low for the remaining 1,410.

50/50 magnitude PWM output is used with two different types of amplifiers. When driving a brushless PM (permanent magnet) motor the magnitude signal is connected to a half bridge driver. When driving a DC brushed motor an H-bridge type amplifier is used however the magnitude signal of the H-bridge is always turned on, and the magnitude output of the chipset is connected to the sign input of the H-bridge. This alternative method of controlling an H-bridge is occasionally useful in situations where motor back-EMF during deceleration is a problem using the standard sign magnitude schemes.

9.4.3 16-Bit DAC

In this mode the motor command for a given phase is output directly to the chipset's peripheral bus where it is assembled into an analog voltage using a DAC (Digital to Analog Converter). The CP chip writes the DAC output value to each enabled channel in this mode at the cycle frequency. So (for example) for a MC2140 which has all four axes set to DAC mode and which is operating at the standard cycle time of 400 µsec, each axis will be written to once per 400 µsec, with the writes separated by 100 µsec

For one or two phase motors, one DAC output is used for each phase. For three phase motors, only two DAC outputs are used, the third phase will always be an analog signal equal to –1 * (P1 + P2) where P1 is the output for phase 1 and P2 is the output for phase 2. If necessary this third phase signal may be realized using an inverting summing amplifier in the external circuitry. Generally though this is not necessary since the majority of 3-phase off-the-shelf amplifiers accept two phases and internally construct the third.

The output value written has a resolution of 16 bit. This value is offset by 8000h, so a value of 0 will correspond to the most negative output. A value of 8000h corresponds to zero output, and a value of 0FFFFh corresponds to the most positive output.

DACs with resolutions lower than 16 bits may be used. In this case output values must be scaled to the high-order bits of the 16-bit data word. For example, to connect to an 8-bit DAC, pins Data8-15 are used. The contents of the low-order 8 bits (Data0-7) are transferred to the data bus, but ignored.

The chipset writes the DAC values using the external peripheral bus described in Section 9.3.

The addresses that the chipset writes to are shown in the table below:

Address	Motor / phase
4000h	Axis 1, phase 1
4001h	Axis 1, phase 2
4002h	Axis 2, phase 1
4003h	Axis 2, phase 2
4004h	Axis 3, phase 1
4005h	Axis 3, phase 2
4006h	Axis 4, phase 1
4007h	Axis 4, phase 2

For more information on DAC signal timing and conditions, see the DAC pin descriptions and peripheral write interface timing diagram in the Technical Specifications document for your chipset.

10 Host Communication

The Navigator Motion Processor communicates with its host(s) through either of two ports: a bi-directional parallel port, or an asynchronous serial port. The two ports may be connected to the same or different host processors, as shown in the diagram below.

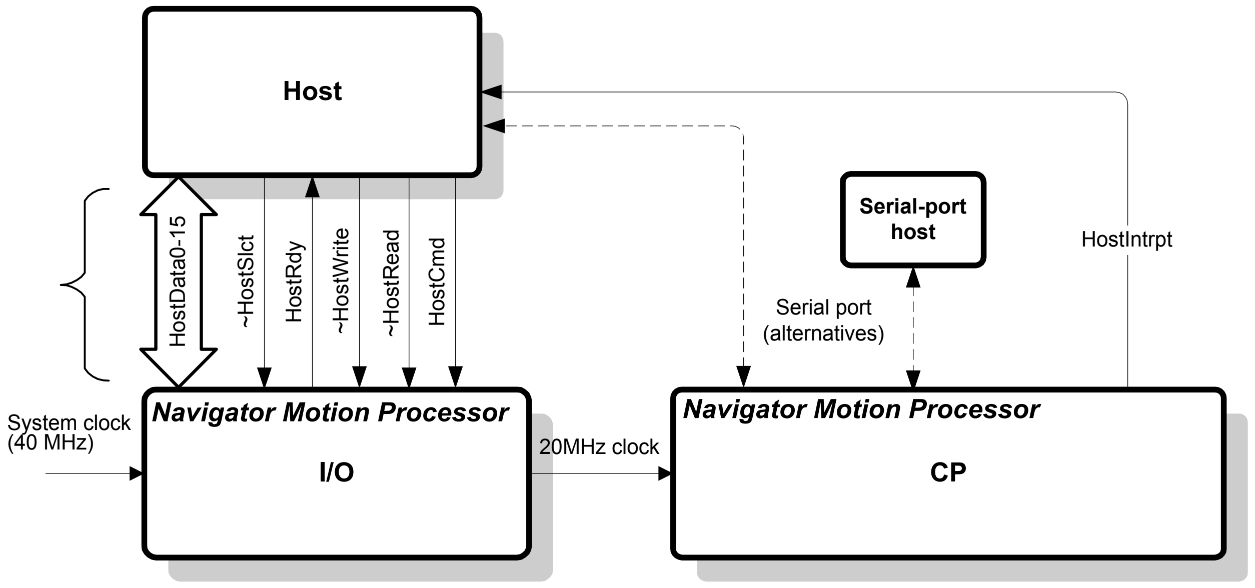


Figure 10-1. Host-Motion Processor communications

The chipset accepts commands from the host in a packet format. By sending sequences of commands the host can control the behavior of the motion system as desired, and monitor the status of the chipset and the motors.

10.1 Primary and Diagnostic Ports

In a typical motion system only one host communication method will be used at a time to control the motion system: parallel (bi-directional communication data bus), or serial (asynchronous serial port).

In some cases however it is advantageous to control the motion system through one channel of communication while monitoring through another. In such a case, when both parallel and serial interface ports are used, one is designated the *primary* port, and the other is designated the *diagnostic* port. The primary port may be used for all host-chipset communications and control. The diagnostic port is normally used to monitor performance. By default only a limited set of host instructions can be issued through it.

At power-up, the parallel port, if it is present, is made primary by default, the serial port becoming the diagnostic port. If the parallel port is disabled the serial port is primary.

10.1.1 Diagnostic port functions

As its name implies, the diagnostic port is used to keep tabs on the performance of the system. By default, the diagnostic port is only allowed to issue commands that do not affect the internal state of the chipset. These commands include all of the Get commands. In addition, the diagnostic port has access to all commands used to take data traces (including the SetTrace commands and the SetBufferReadIndex command).

To override these restrictions, the command SetDiagnosticPortMode command may be issued by the primary port. This allows the diagnostic port to be granted full access to the chipset's command set. Caution should be exercised however because if both the primary and diagnostic port issue motion commands, unexpected behavior may result.

The diagnostic port is especially useful because the primary port need not be aware that a diagnostic port is attached. For example during system development, the user may want to conveniently "look inside" the chipset and monitor servo performance using the trace facility. Rather than write all of this code into the user's application the user can access a separate system connected to the diagnostic port (typically the serial port) which executes a software program that can perform trace and display functions.

10.2 Parallel Communication Port

The bi-directional parallel port is configured to operate in one of two modes, as follows:

16-bit mode	The motion processor transfers instructions and data as full 16-bit words, using the entire 16-bit data path.
8-bit mode	The motion processor transfers instructions and data as full 16-bit words, however using an 8-bit data path. Words are transferred in two successive bytes; the high-order byte of each word is transferred first in all cases. This mode allows access to all features of the Navigator instruction set even when the host is limited to an 8-bit data path.

The parallel port configuration is determined by the HostMode1 and HostMode0 pins on the I/O chip, which must be set as follows:

I/O Mode	HostMode1	HostMode0
16/16 mode	0	0
1st-gen compatible	0	1
8/16 mode	1	0
Serial port	1	1

If the parallel port is not used, HostMode1 and HostMode0 must both be set to 1. This disables the parallel port and makes the serial port the default primary port (see Section 10.1).

In both the 16/16 and 8/16 parallel communication modes, a chip command consists of a 16-bit word with the axis number contained in the most significant byte and the command operand code contained in the least significant byte. In the 1st generation compatible mode a single 8-bit byte is transferred, containing only the command operand code.

10.2.1 Control signals

Five control signals synchronize communications through the parallel port[1]: ~HostSlct, HostRdy, ~HostWrite, ~HostRead, and HostCmd.

Signal	Description		
~HostSlct	Set by host - when this signal is asserted (low), the host parallel port is selected for operations.		
HostRdy	Set by chipset - when high, indicates to the host that the motion processor's host port is available for operations.		
~HostWrite	Set by host - when asserted low, allows a data transfer from the host to the chipset.		
~HostRead	Set by host - when asserted low, allows a data word to be read by the host from the chipset.		
HostCmd	Used in conjunction with the ~HostRead and ~HostWrite signals as follows:		
	When	*and HostCmd is*	
	~HostWrite is low	low:	write data word to chipset
		high:	write instruction word to chipset
	~HostRead is low	low:	read data word from chipset
		high:	read status byte from chipset (see Section 10.2.3)

10.2.2 Parallel port I/O operations

Using the five parallel port control signals of ~HostSlct, HostRdy, ~HostWrite, ~HostRead, and HostCmd it is possible to perform all necessary operations to send commands to the chipset.

There are three operations that accomplish this, the *instruction word write*, the *data word write*, and the *data word read*. By performing these operations in the correct sequence complete command packets can be assembled and sent to the chipset. Command format is discussed in an upcoming section.

instruction word write - In 16-bit bus mode this is accomplished by asserting ~HostSlct and ~HostWrite low, asserting HostCmd high, and loading the data bus with the desired 16-bit instruction word value. In 8-bit bus mode the control signals are the same except only the low 8 bits of the data bus hold data, and the operation is performed twice. On the first 8-bit write the data bus should contain the high byte of the instruction word. On the second 8-bit write the data should contain the low byte of the instruction word.

data word write - In 16-bit bus mode this is accomplished by asserting ~HostSlct, and ~HostWrite low, asserting HostCmd low, and loading the data bus with the desired 16-bit data word value. In 8-bit bus mode the control signals are the same except only the low 8 bits of the data bus hold data, and the operation is performed twice. On the first 8-bit write the data bus should contain the high byte of the data word. On the second 8-bit write the data should contain the low byte of the data word.

data word read - In 16-bit bus mode this is accomplished by asserting ~HostSlct, and ~HostRead low, asserting HostCmd low, and storing the value asserted by the chipset on the 16-bit data bus. In 8-bit bus mode the control signals are the same except only the low 8 bits of the data bus hold data, and the operation is performed twice. On the first 8-bit read the data bus will contain the high byte of the data word. On the second 8-bit read the data will contain the low byte of the data word.

[1] A sixth control signal, HostIntrpt, is connected to the CP chip. It is not used directly in communications; see Section 6.7 for discussion.

At the beginning of each of these operations the ~HostSlct signal must be low. This indicates that the chipset is ready to receive or transmit a new data or instruction word. In between 8-bit transfers the ~HostRdy does not need to be checked. For example after checking the ~HostRdy for the high byte of any of these 2-byte transfers (*instruction word write, data word write, or data word read*) the ~HostRdy does not have to be checked again to transfer the low byte.

For more detailed electrical information on these operations, see the pin descriptions and timing diagrams in the *Technical Specifications* document for your chipset.

Before any parallel host I/O operation is performed, the user must make sure that the ~HostRdy signal is high (chipset ready). After each word (instruction or data) is read or written, this signal will go low (chipset busy). It will return to ready when the chipset is ready to transfer the next word.

10.2.3 The *status read* operation

There is a special operation called a status read which is not directly related to reading or writing to the chipset. The status read allows the user to determine the state of some of the chipset's host interface signals and flags without having to develop special decode logic.

A status read operation is performed by asserting ~HostRead and ~HostSlct low, HostCmd high, and reading the data bus. The resultant data word sent by the chipset contains the following information:

Bit number	Description
0-12	unused, set to 0
13	Holds value of HostIOError signal (see section 10.3.3 for more information on this signal).
14	Holds value of HostIntrpt signal. A 1 indicates the signal level is high.
15	Holds value of HostRdy signal. 1 indicates the signal level is high.

When communicating in 8/8 mode, these bits are returned in a single byte as follows

Bit number	Description
0-4	unused, set to 0.
5	Holds value of HostIOError signal (see section 10.3.3 for more information on this signal).
6	Holds value of HostIntrpt signal. 1 indicates the signal level is high.
7	Holds value of HostRdy signal. 1 indicates the signal level is high.

Status reads can be performed at any time, regardless of the state of the HostRdy signal.

10.3 Parallel Host I/O Commands

Each command sent by the host has an overall format which does not change, even if the amount of data and nature of the command varies somewhat. This generic format is as follows:

Each command has an instruction word (16 bits) which identifies the command. Then there may be zero or more words of data associated with the command that the host writes to the chipset, followed by zero or more words of data that the host reads back from the chipset. Finally there is an optional checksum that may be read by the host to check that communications are occurring properly. Whether there is data written to the chipset and data written to the host depends on the type of command.

10.3.1 Packet Format

All communications to/from the chip set take the form of packets. A packet is a sequence of transfers to/from the host resulting in a chip set action or data transfer. Packets can consist of a command with no data (Dataless Command), a command with associated data that is written to the chip set (Write Command) or a command with associated data that is read from the chip set (Read Command).

All commands with associated data (read or write) have 1,2 or 3 words of data. See the *Navigator Programmer's reference* for more information on the length of specific commands.

If a read or a write command has 2 words of associated data (a 32 bit quantity) the high word is loaded/read first, and the low word is loaded/read second.

The following charts show the generic command packet sequence for a Dataless Command, a Write Command, and a Read Command. The hardware communication operation described in section 10.2.1 to accomplish each type of transfer is shown in the left column.

```
                              Dataless Command

Time            →       →       →       →

Cmd Write:  Cmd word
Data Write:
Data Read:           [packet checksum]
```

```
                              Write Command

Time            →       →       →       →

Cmd Write:  Cmd word
Data Write:            word 1 [word 2]  [word 3]
Data Read:                                         [packet checksum]
```

```
                              Read Command

Time            →       →       →       →

Cmd Write:  Cmd word
Data Write:            [word 1]
Data Read:                      word 1 [word 2] [packet checksum]
```

[ ] Indicates an optional operation

10.3.2 Checksum

It is possible to retrieve a checksum at the end of each command, whether the command is a read command or a write command. The checksum can enhance reliability for critical applications, particularly in very noisy electrical environments, or when the communication signals are routed over a medium that may have data losses. The checksum consists of the low-order 16 bits of the sum of all preceding words transmitted in the command. For example if a SetVelocity instruction (which takes two 16-bit words of data) is sent with a data value of FEDCBA98 (hex), the checksum would be:

`0011h`	code for SetVelocity instruction
`+ FEDCh`	first data word
`+ BA98h`	second data word
`1B985h`	checksum = B985 (low-order 16 bits only)

Reading the checksum is optional. Recovery from an incorrect command transfer (bad checksum) will depend on the nature of the packet. Buffered operations can always be re-transmitted, but a non-buffered instruction (one that causes an immediate action) might or might not be re-transmitted, depending on the instruction and the state of the axis.

10.3.3 Host I/O Errors

There are a number of checks that the chipset makes on the command sent to the chipset. These checks improve safety of the motion system by eliminating some obviously incorrect command data values. All such checks associated with host I/O commands are referred to as Host I/O errors. If any such error occurs, bit 13 of the I/O status read word (see previous section for definition of this word) is set. To determine the cause, the command GetHostIOError is used. Executing the GetHostIOError command also clears both the error code and the I/O error bit in the I/O status read word.

I/O errors codes which are returned by the GetHostIOError command are as follows:

Code	Indication	Cause
0	No error	No error condition.
1	Navigator reset	Default value of error code on reset or power-up.
2	Invalid instruction	Instruction is not valid in the current context, or an illegal instruction code has been detected.
3	Invalid axis	The axis number contained in the upper bits of the instruction word is not supported by this chipset.
4	Invalid parameter	The parameter value sent to the motion processor was out of its acceptable range.
5	Trace running	An instruction was issued that would change the state of the tracing mechanism while the trace is running. Instructions which can return this error are SetTraceVariable, SetTraceMode & SetTracePeriod.
6	Reserved	
7	Block bound exceeded	1. The value sent by SetBufferLength or SetBufferStart would create a memory block which extends beyond the allowed limits of 400h - 7FFFFFFFh. 2. Either SetBufferReadIndex or SetBufferWriteIndex sent an index value greater than or equal to the block length.
8	Trace zero	SetTraceStart Immediate was issued, but the length of the trace buffer is currently set to zero.

Code	Indication	Cause
9	Bad checksum	*(Serial port only)* The checksum complied and returned by Navigator does not match that sent by the host.
Ah	Not primary port	A prohibited instruction (one which can be executed only through the primary port) was issued through the diagnostic port.
Bh	Negative velocity	An attempt was made to set a negative velocity without the axis being in velocity contouring profile mode.
Ch	S-curve change	The axis is currently executing an S-curve profile move and an attempt was made to change the profile parameters. This is not permitted.
Dh	Limit event pending	A limit switch event has occurred.
Eh	Move into limit	An attempt was made to execute a move without first clearing the limit bit(s) in the Event Status register.

10.4 Serial Port

In addition to the parallel interface, the Navigator motion processor provides an asynchronous serial connection. This serial port may be configured to operate at baud rates ranging from 1200 baud to 416,667 baud, and may be used in point-to-point or multi-drop mode.

10.4.1 Configuration

After reset, the chipset reads a 16-bit value from its peripheral bus (location 200h) which it uses to set the default configuration of the serial port.   If the serial port is to be used, then external hardware should be used to decode this access and provide a suitable configuration word as described below. See section 9.3 for details on peripheral bus I/O.

Alternatively, if adding external-decoding hardware is not desirable then the CP's external data bus may be pulled up using high value resisters (for example 10 Kohm). This will cause the chipset to read the value 0FFFFh from address 200h. When this value is read the chipset will setup the serial port in a default configuration of 9600 baud, no parity, one stop bit, point-to-point mode.

The configuration word read by the CP at address 200h is organized as follows:

Bit	Parameter	Indications			
0-3	Transmission rate selector	0h	1200 bits per second	4h	57600 bps*
		1h	2400 bps	5h	115200 bps*
		2h	9600 bps	6h	250000 bps
		3h	19200 bps	7h	416667 bps
4-5	Parity selector	0h	None		
		1h	Odd parity		
		2h	Even parity		
		3h	Reserved (do not use)		
6	Number of stop bits	0h	1 stop bit		
		1h	2 stop bits		
7-8	Protocol type	0h	Point-to-point		
		1h	Reserved (do not use)		
		2h	Multi-drop (address bit mode)		
		3h	Multi-drop (idle line mode)		
9-10	*Reserved*				
11-15	Multi-drop address selector. Should be zero in point-to-point mode.	0h	Address 0		
		1h	Address 1		
		…			
		31h	Address 31		

*To insure synchronization between characters at these transmission rates, use 2 stop bits.

10.4.2 Control signals

Three signals, SrlXmt, SrlRcv, and SrlEnable mediate serial transfers. SrlXmt encodes data transmitted from the CP to the host processor. SrlRcv receives data sent from the host to the CP. SrlEnable goes active (high) when data is being transmitted in multi-drop mode. This signal may be connected to the output enable pin of a serial buffer IC to allow tri-stating of the transmit line of a serial bus when not in use. In point-to-point mode the SrlEnable pin is always active (high).

The basic unit of serial data transfer (both transmit and receive) is the byte. Each byte of data consists of the following parts:

One start bit
Eight data bits
An optional even/odd parity bit
One or two stop bits
An extra bit which distinguishes between address bytes and data bytes (address-bit style multi-drop mode only)

10.4.3 Command format

The command format used to communicate between the host and chipset consists of a command packet sent by the host processor followed by a response packet sent by the chipset.

Command packets sent by the host contain the following fields:

Field	Byte#	Description
Address	1	One byte identifying which CP the command packet is being sent to. This field should always be zero in point-to-point mode.
Checksum	2	One byte value used to validate packet data. See description below.
Instruction code	3-4	Two byte instruction, sent upper byte (axis number) first. The command codes are the same as used in the parallel communication mode.
Data	5-	Zero to six bytes of data, sent most significant byte first. See the individual command descriptions for details on what data is required for each command.

In response to the command packet, the chipset will respond with a packet of the following format:

Field	Byte#	Description
Status	1	Zero if the command was completed correctly, otherwise an error code specifying the nature of the error.
Checksum	2	A one-byte checksum value used to validate the packet's integrity. See details below.
Data	3-	Zero to six bytes of data. No data will be sent if an error occurred in the command (i.e. the status byte was non-zero). If no error occurred, then the number of bytes of data returned would depend on what command had the CP was responding to. Data is always sent MSB first.

10.4.3.1 Checksums

Both command and response packets contain a checksum byte. The checksum is used to detect transmission errors, and allows the chipset to identify and reject packets which have been corrupted during transmission or which were not properly formed.

Unlike the parallel interface, checksums are mandatory when using serial communications. Any command packets sent to the chipset containing invalid checksums will not be processed and will result in a data packet being returned containing an error status code.

The serial checksum is calculated by summing all bytes in the packet (not including the checksum) and negating (i.e. taking the two's complement of) the result. The lower eight bits of this value are used as the checksum. To check for a valid checksum, all bytes of a packet should be summed (including the checksum byte) and if the lower eight bits of the result are zero then the checksum is valid.

For example, if a command packet is sent to chipset address 3, containing command 0177h (SetMotorCommand for axis 2) with the one word data value 1234h, then the checksum will be calculated by summing all bytes of the command packet (03h + 01h + 77h + 12h + 34h = C1) and negating this to find the checksum value (3Fh). On receipt, the CP will sum all bytes of the packet and if the lower eight bits of the result are zero then it will accept the packet (03h + 3Fh + 01h + 77h + 12h + 34h = 100h).

10.4.3.2 Transmission protocols

The Navigator chipsets support the ability to have more than one chipset on a serial 'bus', thereby allowing a chain, or network of chipsets to communicate on the same serial hardware signals.

There are three methods supported by the serial port to resolve timing problems, transmission conflicts, and other issues that may occur during serial operations. These are point-to-point (used when there is only one device connected to the serial port), multi-drop address bit-mode (used when there are multiple devices on the serial bus), and multi-drop idle-line mode (also used when there are multiple devices on the serial bus). The next sections describe these three transmission protocols.

10.4.3.3 Point-to-point mode

Point-to-point serial mode is intended to be used when there is a direct serial connection between one host and one chipset. In this mode the address byte is not used by the CP (except in the calculation of the checksum), and the chipset responds to all commands sent by the host.

When in point-to-point mode there are no timing requirements on the data transmitted within a packet. The amount of data contained in a command packet is determined by the command code in the packet. Each command code has a specific amount of data associated with it. When the CP receives a command code it waits for all data bytes to be received before processing the command. The amount of data returned from any command is also determined by the command code. After processing a command the chipset will return a data packet of the necessary length.

When running in point-to-point mode there is no direct way for the chipset to distinguish the beginning of a new command packet, except by context. It is therefore important for the host to remain synchronized with the chipset when sending and receiving data. To ensure that the processors stay in synchronization it is recommended that the host processor implement a time limit when waiting for data packets to be sent by the chipset. The minimum timeout period suggested is the amount of time required to send one byte at the selected baud rate plus one millisecond. For example, at 9600 baud each bit takes 1/9600 seconds to transfer, and a typical byte consists of 8 data bits, 1 start bit, and 1 stop bit. Therefore, one byte takes just over 1 millisecond, and the recommended minimum timeout is 2 milliseconds.

If the timeout period elapses between bytes of receive data while the host is waiting on a data packet, then the host should assume that it is out of synchronization with the chipset. To resynchronize, the host should send a byte containing zero data and wait for a data packet to be received. This process should be repeated until a data packet is received from the chipset, at which point the two processors will be synchronized.

10.4.3.4 Multi-drop protocols

Two multi-drop serial protocols are supported by the chipset. Multi-drop modes are intended to be used on a serial bus in which a single host processor communicates with multiple chipsets (or other subordinate devices). In this mode the address byte which starts a command packet is used to indicate which device the packet is intended for. Only the addressed device will respond to the packet. Therefore, it is important to properly set up the chipset address (using the serial configuration word described above), and to include this address as the first byte of any command packet destined for the chipset.

Because the address that starts a command packet is used to enable or disable the response from a chipset in multi-drop mode, the multi-drop protocols must include a method to avoid losing synchronization between the host and the chipset(s) because it would be difficult to regain synchronization in this environment. The two multi-drop protocols differ in the methods they use to control packet synchronization.

Note that the multi-drop protocols may also be used when the host and chipset are wired in a point-to-point configuration as long as the host always transmits the correct address byte at the start of a packet, and follows any additional rules detailed below for the selected protocol. This mode of operation allows the host to be sure that it will remain synchronized with the chipset without implementing the timeout and re-synch procedure outlined above.

10.4.3.5 Idle-line mode

When the idle-line protocol is used, the chipset imposes tight timing requirements on the data sent as part of a command packet. In this mode, the chipset will interpret the first byte received after an idle period as the start of a new packet. Any data already received will be discarded.

The timeout period is equal to the time required to send ten bits of serial data at the configured baud rate (roughly 1 millisecond at 9600 baud for example). If a delay of this length occurs between bytes of a command packet then the bytes already received will be discarded, and the first character received after the delay will be interpreted as the address byte of a new packet.

10.4.3.6 Address bit mode

When the address bit protocol is used, each byte of data transmitted or received by the chipset contains an extra bit of data (just after the last data bit). This bit is used to identify the address byte of a packet. Any byte received by the chipset which has this bit set will be interpreted as the start of a new command packet. If the chipset was in the process of receiving data from an earlier command packet then that data will be discarded.

11 Using External Memory

11.1 Memory configuration

The Navigator chipset is capable of accessing external memory used to store trace data. In addition it is possible to access the external memory independent of the trace function, thereby allowing it to be used for generic storage such as for product configuration information.

If external RAM is present, it must be configured with a width of 16 bits. Any size of RAM may be supported from 1K and up although the first 1024 words of external memory are always reserved for the use of the chipset. The total external memory available for functions such as trace must therefore be calculated accordingly.

The chipset address and data busses are used to interface to the external memory. The chipset uses 15 address lines for external memory operation that allow it to directly access up to 32K words of memory. To allow a greater addressable space, the CP also makes use of a memory page selector (mapped into peripheral address space at location 2000h) which allows up to 64K pages to be accessed. This increases the maximum addressable memory space to 2,048M words (2,147,483,648 words, each 16 bits wide).

This memory organization gives the designer several options as to the level of support provided to this feature:

Memory size	Notes
None	The external memory space is optional. Diagnostic trace and other features that make use of the external memory space cannot be used.
1K to 32K	The first 1K (1024 words) are reserved by the CP. All memory fits into the first page, so there is no need to decode writes to the external page pointer.
More then 32K	An external page pointer must be used. The first 1K words of page zero are reserved for use by the CP.

11.1.1 Memory page pointer

If a memory address space of more then 32K words is desired, then an external page register must be supported.

Before the CP writes to external memory space, it writes a 16-bit page number to peripheral address location 2000h (see section 8.3 for more details on peripheral I/O). This value should be latched and used as the upper 16 bits of the 31 bit 'true' memory address.

The chipset will never attempt to read from this page number register, so there is no need to provide support for reads. Although the value written by the chipset is 16 bits, it is not necessary to latch any bits that will not be used. For example, if 128K words of memory mapped to the physical addresses 0 – 1FFFFh, then 4 pages of memory (4 x 32K = 128K) will be needed. This corresponds to the low two bits of the page register. When the CP writes a 16-bit value to peripheral address 2000h the lowest two bits should be latched and used as address bits 15 and 16 when accessing external memory. The upper 14 bits of the page register value need not be saved since they will not be used.

11.1.2 External memory signal decoding

The Navigator chipsets support an external RAM memory bus to read and write to external memory.

The bus consists of 4 control signals; ~RAMSlct, W/~R, R/~W, and ~WriteEnbl. In addition there are 15 address lines (Addr0-Addr14) and 16 data lines (Data0-Data15).

11.1.3 External memory read

To perform an external memory read the chipset first loads the memory page pointer using the peripheral bus as described above. Then to perform the read the chipset will drive to a low level the ~RAMSlct signal, the ~Strobe signal, and the W/~R signal. In addition the chipset will drive the address bus high or low based on the specific memory being fetched.

After the Navigator asserts these signals it is expected that the user's peripheral circuitry will assert the requested data onto the chipset's data bus data0-15.

The specific timing for the external memory read function is detailed in the Navigator Technical Specifications for your chipset.

11.1.4 External memory write

To perform an external memory write the chipset first loads the memory page pointer using the peripheral bus described above. Then to perform a write the chipset will drive to a low level the ~RAMSlct signal, the ~Strobe signal, the W/~R signal, and the ~WriteEnbl signal. In addition the chipset will drive the address bus high or low based on the specific memory being written to and it will also assert the data word on to the data bus for the value being written by the chipset.

After the Navigator asserts these signals it is expected that the user's peripheral circuitry will read in and store the data written by the chipset.

The specific timing for the external memory write function is detailed in the Navigator Technical Specification manual.

11.1.5 External memory buffers

The Navigator chipset provides a number of commands that may be used to access the external memory space. Through these commands, up to 16 memory *buffers* may be defined. A buffer describes a contiguous block of memory by defining a base address for the block and a block length. Once a buffer's base address and length have been defined, data values may be written to and read back from the buffer.

When defining memory buffers, the external memory space is treated as a sequence of 32-bit memory locations. Each 32-bit value takes up two 16-bit memory locations in physical memory. Buffer base addresses and lengths both deal with 32 bit quantities, and therefore must be doubled to get the corresponding physical addresses.

When defining memory buffers, the chipset will allow any values to be used for the base address and length as long as those values result in legal addresses. Legal memory addresses range from 200h (corresponding to physical location 400h that falls just after the 1K reserved block) to 3FFFFFFFh (corresponding to physical address 7FFFFFFEh). Unless the full two Gigawords of physical memory are present, it is possible to map a buffer to a memory location that does not contain physical memory.

Accessing such a memory location will result in unpredictable behavior, and should be avoided.

In addition to the base address and length, each memory block maintains a *read index* and a *write index*. The read index may be assigned a value between 0 and L-1 (where L is the buffer length). It defines the location from which the next value will be read. Similarly, the write index ranges from 0 to L-1 and defines the location at which the next value will be written. When a value is read from the memory buffer the read index is automatically incremented, thereby selecting the next value for reading. The write index is incremented whenever a value is written to a buffer. If either index reaches the end of the buffer, it is automatically reset to 0 on the next read/write operation.

11.1.6 External memory commands

This section details host I/O commands which setup, access, and monitor the external memory.

SetBufferStart *bufferID*, *address*

Sets the base address of a buffer. ***bufferID*** is a 16-bit integer in the range 0–15 which specifies which buffer to modify; ***address*** is a 32-bit integer in the range 200h to 3FFFFFFFh which defines the new base address of the buffer.

The chipset adds ***address*** to the current buffer length (as set by the SetBufferLength instruction) to be sure that the buffer will not extend beyond the addressable memory limit. If it would extend beyond the limit, the instruction is ignored and the instruction error bit is set.

GetBufferStart *bufferID*

Returns the base address of the specified buffer.

SetBufferLength, *bufferID*, *length*

Sets the length of a buffer. ***bufferID*** is a 16-bit integer in the range 0–15. ***length*** is a 32-bit integer in the range 1 to 3FFFFFFFh.

The chipset adds ***length*** to the current buffer base address (as set by the SetBufferStart instruction) to be sure that the buffer will not extend beyond the addressable memory limit. If it would, the instruction is ignored and the instruction error bit is set.

SetBufferStart and SetBufferLength reset the buffer indexes to zero.

GetBufferLength *bufferID*

Returns the length of the specified buffer.

SetBufferReadIndex *bufferID*, *index*

Sets the read index for the specified buffer. ***index*** is a 32-bit integer in the range 0 to ***length***-1, where ***length*** is the current buffer length. If ***index*** is not in this range, it is not set, and an instruction error is generated.

GetBufferReadIndex *bufferID*

Returns the value of the read index for the specified buffer.

SetBufferWriteIndex *bufferID*, *index*

Sets the write index for the specified buffer. ***index*** is a 32-bit integer in the range 0 to ***length***-2, where ***length*** is the current buffer length. If ***index*** is not in this range, it is not set, and an instruction error is generated.

GetBufferWriteIndex *bufferID*

Returns the value of the write index for the specified buffer.

ReadBuffer *bufferID*

Returns a 32-bit value from the specified buffer. The location from which the value is read is determined by adding the base address to the read index. After the value has been read, the read index is incremented. If the result is equal to the current buffer length, the read index is set to zero.

WriteBuffer *bufferID, value*

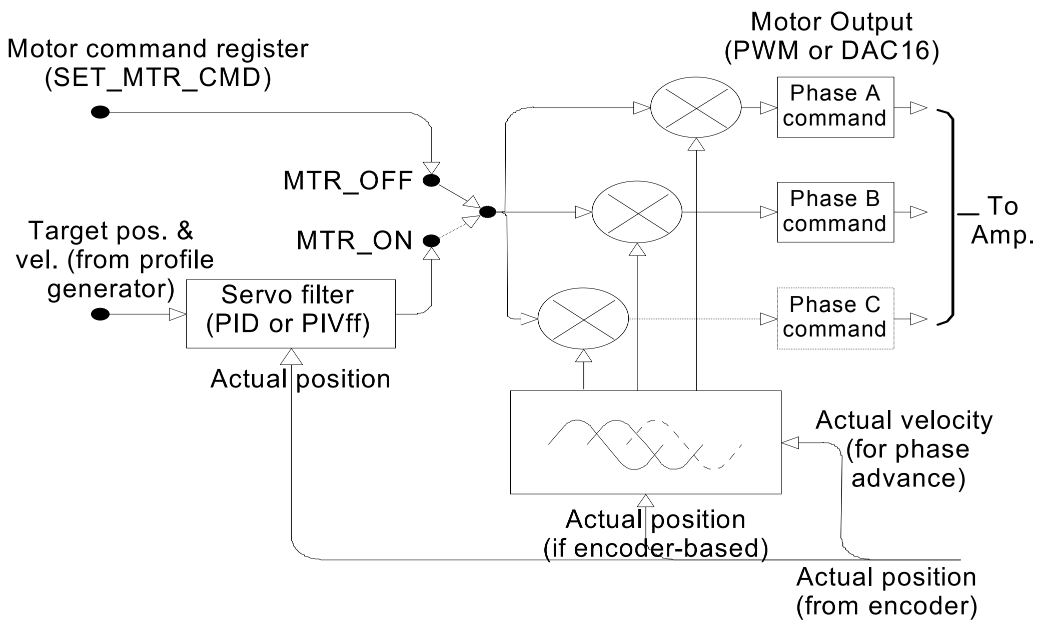
Writes a 32-bit value to the specified buffer. The location to which the value is written is determined by adding the base address to the write index. After the value has been written, the write index is incremented. If the result is equal to the current buffer length, the write index is set to zero.

12 Sinusoidal Commutation (MC2300, MC2800)

12.1 Overview

In addition to trajectory generation and servo loop closure the MC2300 and MC2800 chipsets provides sinusoidal motor commutation of 3 and 2-phase brushless motors. The MC2800 can also provide single-phase output, for example to control a brushed motor. This allows for mixed motor combinations using just one chipset.

The following diagram shows an overview of the control flow of the sinusoidal commutation portion of the MC2300 chipset:



The commutation portion of the chipset uses as input the motor command signal from either the servo filter or the motor command register (depending on whether the chipset is in closed loop or open loop mode). This pre-commutated command signal is then multiplied by commutation values derived from an internal lookup Sin/Cos table.

The commutation angle used in the Sin/Cos lookup is determined by the position encoder as well as parameters set by the host processor which relate the specific encoder used to the motor magnetic poles.

Two commutation waveforms are provided, one appropriate for 3-phase devices with 120 deg. separation between phases (such as brushless motors), and one appropriate for 2-phase devices with 90 deg. separation between phases (such as stepper motors).

Other features of the MC2300 chipset are the ability to use Hall-sensor inputs for phase initialization, to use an index pulse to maintain commutation synchronization, to pre-scale the encoder input to support a wider variety of feedback devices, and to provide velocity-based phase advance for smoother and more efficient high speed operation.

12.2 Selecting single-phase output with the MC2800

To operate a brushed single phase motor with the MC2800, the command SetNumberPhases is used. Additionally the output mode should be set according the requirements as shown in the following example:-

Command	Description
SetOutputMode m	Set the output mode
SetNumberPhases 1	Set the number of phases to one

12.3 Commutation Waveforms

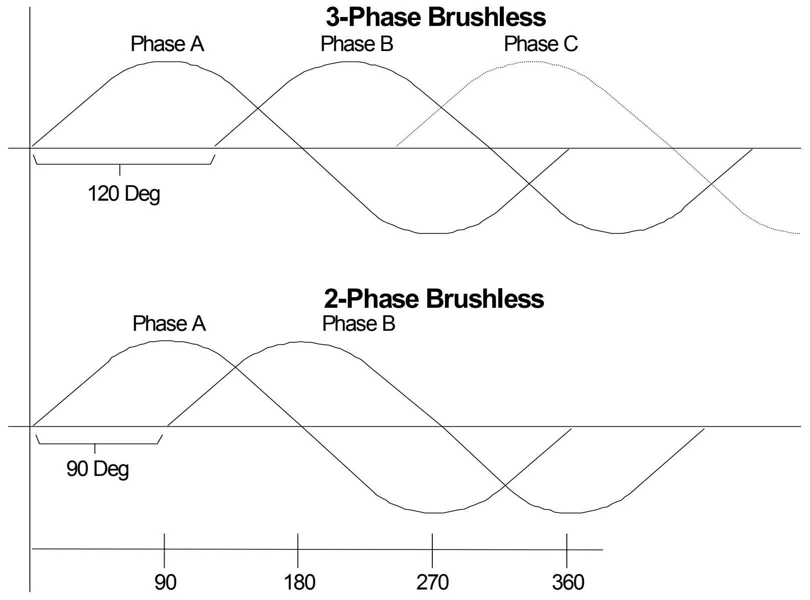
The MC2300 supports two commutation waveforms, a 120 degree offset waveform appropriate for 3-phase brushless motors, and a 90-degree offset waveform appropriate for 2-phase brushless motors. To specify the 3-phase brushless waveform the command SetNumberPhases 3 is used, and to set it for 2-phase brushless motors the command SetNumberPhases 2 is used.

Depending on the waveform selected, as well as the motor output mode selected (PWM or DAC16), either 2 or 3 commutated output signals per axis will be provided by the chipset. The following chart shows this.

Waveform	Motor output mode	Number of output signals & name
3-phase	PWM5050	3 (A, B, C)
3-phase	PWMSign/Mag	2 (A, B)
3-phase	DAC16	2 (A, B)
2-phase	PWM5050	2 (A, B)
2-phase	PWMSign/Mag	2 (A, B)
2-phase	DAC16	2 (A, B)

For specific pin assignments of the PWM and DAC16 motor output signals see the ‘Pin Descriptions' section of the MC2300 Technical Specifications manual.

The diagram below shows the phase A, B, and C commutation signals for a 3-phase brushless motor, and the phase A and phase B signals for a 2-phase brushless motor.



12.4 Commutation Parameters

To perform sinusoidal commutation it is necessary to specify the number of encoder counts per electrical cycle. To determine this value the number of electrical cycles of the motor, along with the number of encoder counts per motor revolution must be known. Knowing these two quantities the number of encoder counts per electrical cycle is given by the following equation:

Counts_per_cycle = Counts_per_rot/Electrical_cycles

where:

Counts_per_rot is the number of encoder counts per motor rotation

Electrical_cycles is the number of motor electrical cycles

The number of electrical cycles can normally be determined by examining the motor manufacturer's specification. The number of electrical cycles is usually half the number of poles. Care should be taken not to confuse poles with pole pairs.

The command used to set the number of encoder counts per electrical cycle is SetPhaseCounts. To read back this value use the command GetPhaseCounts.

12.5 Index Pulse Referencing

To enhance long term commutation reliability the MC2300 provides the ability to utilize an index pulse input from the motor encoder as a reference point during commutation. By using an index pulse during the phase calculations any long term loss of encoder counts which might otherwise affect the accuracy of the commutation are automatically eliminated.

To utilize index pulse referencing the motor encoder chosen must provide an index pulse signal to the chipset once per rotation. This index pulse is connected to the chipset using the Index signal of the I/O chip (see the 'Pin Descriptions' section of the *MC2300 Technical Specifications* manual for more information).

Index pulse referencing is recommended for all rotary brushless motors with quadrature encoders. For linear brushless motors it is generally not used, although it can be used as long as the index pulses are arranged so that each pulse occurs at the same phase angle within the commutation cycle.

When using an index pulse the number of encoder counts per electrical cycle is not required to be an exact integer. In the case that this value is not an integer, the nearest integer should be chosen. Conversely, if index pulses are not being used then the number of counts per electrical cycle must be an exact integer, with no remainder.

For example if a 6-pole brushless motor is to be used with an encoder without an index pulse than an encoder with 1200 counts per rotation would be an appropriate choice, but an encoder with 1024 would not because 1024 cannot be divided by 3 evenly.

The command SetPhaseCorrectionMode is used to enable/disable index pulse phase correction.

Index pulse referencing is performed automatically by the chipset, regardless of the initialization scheme used (algorithmic, Hall-based, microstepping, or direct set).

12.6 Commutation Error Detection

With an index signal properly installed the chipset will automatically correct any small losses of encoder counts that may occur.

If the loss of encoder counts becomes excessive however, or if the index pulse does not arrive at the expected location within the commutation cycle, a "commutation error" is said to occur. The commutation error bit (11) in the Event Status register is set whenever a commutation error occurs. This bit is set if the required correction is greater than (PhaseCounts/128)+4. Commutation errors are caused by a number of circumstances. The most common are listed below:

- noise on the A or B encoder lines
- noise on the index line
- incorrect setting of encoder counts per electrical cycle

For each instance that a commutation error occurs phase referencing will not occur for that index pulse. Depending on the cause of the error the commutation error may be a one-time event, or may occur continuously after the first event.

When a commutation error occurs bit #11 of the EventStatus word is set high (1). This condition can also be used as a source of host interrupts so the host can be automatically notified of a commutation error. To recover from a commutation error this bit is cleared by the host, however depending on the nature of the error it is possible that commutation errors will continue to be generated.

A commutation error may indicate a serious problem with the motion system, potentially resulting in unsafe motion. It is the responsibility of the host to determine and correct the cause of commutation errors.

12.7 Phase Initialization

After startup the chipset must determine the proper commutation angle of the motor relative to the encoder position. This information is determined using a procedure called phase initialization.

The chipset provides four methods to perform phase initialization; algorithmic, Hall Sensor-based, microstepping, and direct-set.

12.7.1 Algorithmic Phase Initialization

To set the chipset for algorithmic initialization use the command SetPhaseInitializeMode and specify Algorithmic as the parameter.

In the algorithmic initialization mode no additional motor sensors beyond the position encoder are required. To determine the phasing the chipset performs a sequence that briefly stimulates the motor windings, and sets the initial phasing using the observed motor response. From the resulting motion the chipset can automatically determine the correct motor phasing.

Depending on the size and speed of the motor, the time between the start of motor phasing and the motor coming to a complete rest (settling time) will vary. To accommodate these differences the amount of time to wait for the motor to settle is programmable using the command SetPhaseInitializeTime. To read back this value use the command GetPhaseInitializeTime.

To minimize the impact on the system mechanics this method utilizes a motor command value set by the host processor to determine the overall amount of power to "inject" into the motor during phase initialization Typically, the amount of power to inject should be in the range of 5 - 25 % of full scale output, but in any case should be at least 3 times the breakaway starting friction. For best results the initialization motor command value can be determined experimentally. The command used to set the motor output level is SetMotorCommand. To read back this value use the command GetMotorCommand.

To execute the initialization procedure, the host command InitializePhase is used. Upon executing this command, the phasing procedure will immediately be executed.

Before the phase initialization command is given however (InitializePhase command), the motor must be turned off (SetMotorMode command), a motor command output must be specified (SetMotorCommand command), and an initialization duration must be specified (SetPhaseInitializeTime command).

During algorithmic phase initialization the motor may move suddenly in either direction. Proper safety precautions should be taken to prevent damage from this movement. In addition, to provide accurate results motor movement must be unobstructed in both directions and must not experience excessive starting friction.

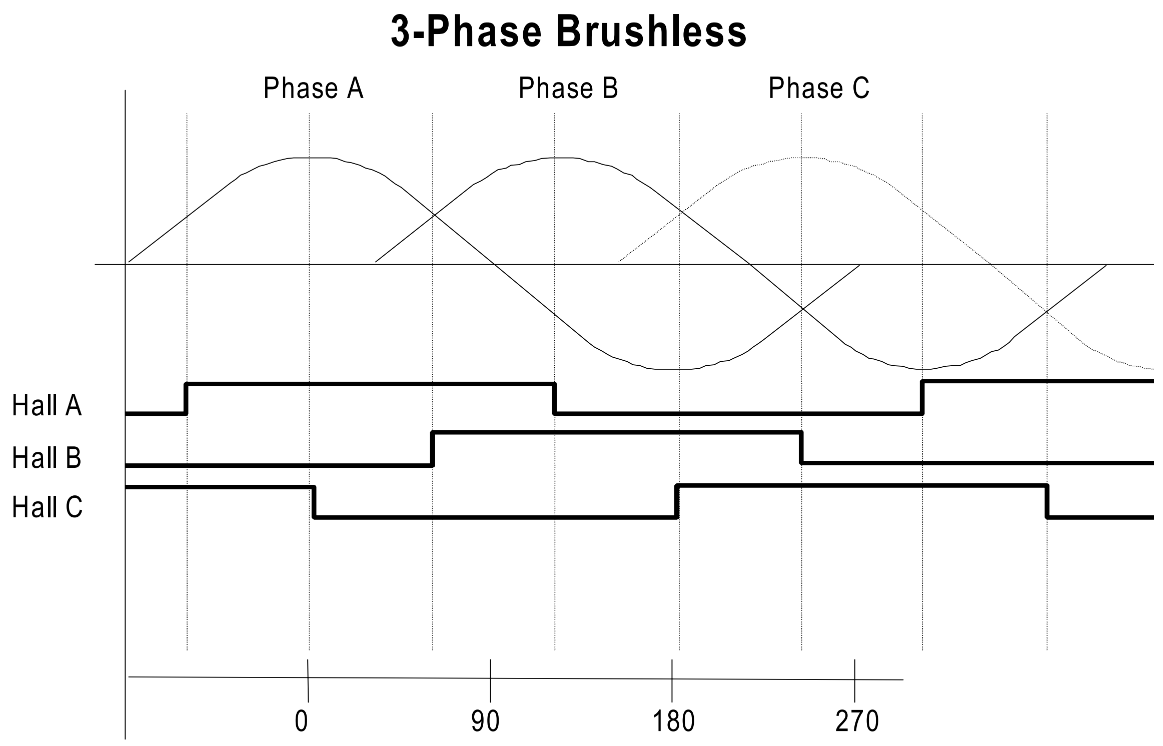
12.7.2 Hall-Based Phase Initialization

To set the chipset for Hall-based initialization use the command SetPhaseInitializeMode and specify Hall-based as the parameter.

In this mode 3 Hall-Sensor signals are used to initially determine the motor phasing, and sinusoidal commutation begins automatically after the motor has moved through one full rotation.

The Hall-Sensor signals are fed back to the chipset through the signals Hall1A-C (axis #1) and Hall2A-C (axis #2), etc. Care should be taken to connect these sensors properly. To read the current status of the hall sensors use the command GetSignalStatus.

The following diagram shows the relationship between the state of the three Hall sensor inputs for each axis and the commutated motor outputs. This graph shows the expected Hall sensor states and winding excitation for forward motion (increasing position).



Unlike the algorithmic method, using Hall-based phase initialization no special motor setup procedures is required. Initialization is performed using the command InitializePhase, and occurs immediately, without any motor motion.

To accommodate varying types of Hall sensors, or sensors that contain inverter circuitry, the signal level/logic interpretation of the Hall sensor input signals can be set through the host.

The command SetSignalSense accepts a bit-programmed word that controls whether the incoming Hall signals are interpreted as active high or active low. To read back this Hall interpretation value use the command GetSignalSense. For details on the programming of this control word see the Navigator Programmer's Reference manual.

Hall-based initialization should only be used with a 3-phase commutation waveform, and with Hall sensors located 120 degrees apart. Hall-sensors located 60 degrees apart should not be used.

12.7.3 Microstepping Phase Initialization

If the location of the index pulse in relation to the motor rotor and case is known then it may be advantageous to use an initialization technique which operates the motor as a microstepper, rotating the motor until the index pulse is found, and then setting the phase angle explicitly.

This scheme is only appropriate for motors which have the index pulse in a fixed and repeatable location within the commutation cycle for all of the motors to be used during manufacturing of the product. Although this is relatively uncommon, it is typical for motors with optical Hall-sensors which use a single disk containing the A, B, index, and Hall sensor information.

To set the chipset for microstepping operation the command SetCommutationMode Microstepping is used. To restore the chipset for encoder-based commutation the command SetCommutationMode Sinusoidal is used. Once the index pulse is encountered the phase angle can be set using the command SetPhaseAngle.

To operate the motor in microstepping mode the motor must be set on (SetMotorMode command), and a motor output value must be provided (SetMotorCommand and Update). In addition the number of encoder counts per electrical cycle should be set to 512.

When in microstepping mode each trajectory 'count' corresponds to 1/256 of a full electrical cycle. For example using a 4-pole motor (2 electrical cycles per motor rotation) a trajectory move of 512 counts will move the motor 1 full motor rotation.

Special care should be taken when initializing the motor using the microstepping method. Because the motor is operated 'open-loop' the resultant coil energization and subsequent rotation may be jerky and abrupt.

Phase initialization using the microstepping method should only be used under special circumstances. It is not generally recommended unless the algorithmic or Hall-based methods cannot be used.

12.7.4 Direct-Set Phase Initialization

If, after power-up, the location of the motor phasing is known explicitly the phase angle can simply be set directly using the SetPhaseAngle command.

This typically occurs when sensors such as resolvers are used where the returned motor position information is absolute in nature (not incremental), and can be used to generate a quadrature data stream as well as be read by the host directly.

12.8 Phase Initialization Programming

The following examples show typical host command sequences to initialize the commutation of a brushless motor for all four initialization methods.

12.8.1 Algorithmic Initialization Sequence

Command	Description
SetOutputMode m	Set the output mode
SetNumberPhases p	Set the number of phases
SetPhaseCounts uuuu	Set # of encoder counts per electrical cycle.
SetPhaseInitializeMode Algorithmic	Set phase initialization method to algorithmic.
SetMotorMode Off	Turn motor off so it doesn't conflict with initialization procedure.
SetPhaseInitializeTime wwww	Set algorithmic phase init duration.

SetMotorCommand yyyy	Set initialization motor command level.
InitializePhase	Perform the initialization.

This sequence will cause the motor to immediately begin the initialization procedure, which will last 'wwww' servo loops long. To determine if the procedure is completed, the command GetActivityStatus can be used. The 'phase initialization' bit will indicate when the procedure is finished. After the initialization procedure is completed the motor should be enabled (SetMotorMode On) if the chipset is to be run in closed loop mode.

12.8.2 Hall-based Initialization Sequence

Command	Description
SetOutputMode m	Set the output mode
SetNumberPhases p	Set the number of phases
SetPhaseCounts uuuu	Set # of encoder counts per electrical cycle.
SetSignalSense vvvv	Set Hall sensor signal interpretation.
SetPhaseInitializeMode Hall	Set phase initialization method to hall based.
InitializePhase	Perform the initialization.

This sequence will cause the chipset to read the Hall sensor signals and initialize the phasing immediately. The motor will not move as a result of this sequence, and no delay is required for further motor operations to be performed.

12.8.3 Microstepping Initialization Sequence

Command	Description
SetOutputMode m	Set the output mode
SetNumberPhases p	Set the number of phases
SetPhaseCounts 512	Set # of encoder counts per electrical cycle to 512 (dec.).
SetCommutationMode Microstepping	Set chipset for microstepping mode.
SetCaptureSource Index	Set capture mode to index (not necessary if already so set).
ResetEventStatus 0	Clear axis status.
GetCaptureValue	Clear out any previous captures.
SetMotorMode On	Turn motor on (not necessary if already on).
SetMotorCommand xxxx	Set motor command value.
SetPosition 560	Set rotation distance a bit more than 1 full motor rotation (assuming 4-pole motor).
SetVelocity yyyy	Set velocity.
SetAcceleration zzzz	Set acceleration.
Update	Start the motion.

This sequence will cause the motor to make a move of somewhat more than 1 rotation. After the Update the host should poll the status word (GetEventStatus) until a capture occurs and then immediately send a SetPhaseAngle command, followed by a SetPhaseOffset command, each loaded with the phase angle required to initialize the phasing.

See the following section of this manual entitled "Adjusting the commutation angle" for more information on determining the correct phase set value.

Once the SetPhaseAngle and SetPhaseOffset commands have been sent by the host the chipset should be initialized for normal commutation operation. This means the phasing mode should be set to encoder-based (SetCommutationMode Sinusoidal) and the correct # of encoder counts per electrical cycle should be set (SetPhaseCounts)

12.8.4 Direct-Set Initialization Sequence

Command	Description
SetOutputMode m	Set the output mode
SetNumberPhases p	Set the number of phases
SetPhaseCounts xxxx	Set the number of encoder counts per electrical cycle (hex).
SetPhaseAngle yyyy	Set phase angle based on information from external sensor.

This sequence will directly set the phase angle to a value determined by another sensor. The set value must be between 0 and the number of encoder counts per electrical cycle.

12.9 Adjusting The Commutation Angle

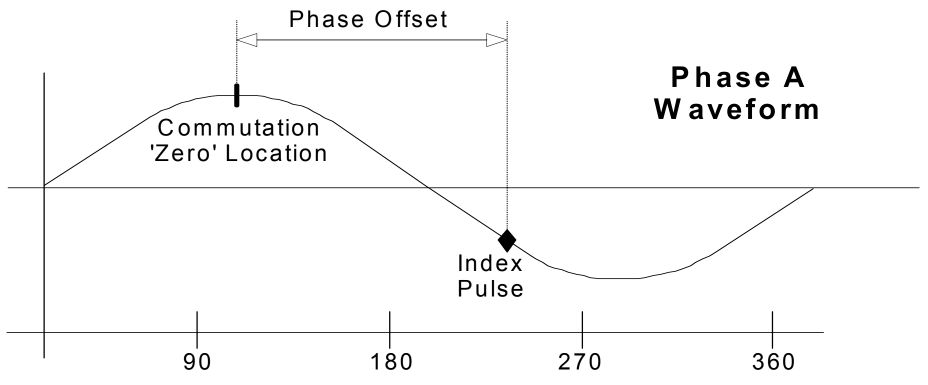
The MC2300 supports the ability to change the motor's commutation angle directly, both when the motor is stationary and when it is in motion. Although this is not generally required it can be useful during testing, or during commutation initialization when the microstepping or direct-set methods are used.

To change the commutation angle when the motor is stationary use the command SetPhaseAngle.

To change the commutation angle while the motor is moving the index pulse is required, and a different command, SetPhaseOffset is used which only takes effect when an index pulse occurs. The following description provides some background on this function.

After phase initialization has occurred the correct commutation angle is stored by the chipset as the offset from the index mark (in encoder counts) to the phase A maximum output value (commutation 'zero' location). This 16-bit offset register can be read using the command GetPhaseOffset.

The following chart shows the relationship between the phase A commutation 'zero' location, the index location, and the phase offset value. For a given motor the index pulse shown in this figure could have been located anywhere within the phase cycle since it will usually vary in position from motor to motor. Only motors that have been mechanically assembled such that the index position is referenced to the motor windings will have a consistent index position relative to the commutation zero location.



Before phase initialization has occurred the phase offset register will have a value of ffff (hex). Once phase initialization has occurred and the motor has been rotated such that at least one index pulse has been received, the phase offset value will be stored as a positive number with a value between 0 and the number of encoder counts per electrical cycle.

To convert the phase offset value, which is in encoder counts, to degrees, the following expression can be used:

$$\text{Offset}_{\text{degrees}} = 360 * \text{Offset}_{\text{counts}}/\text{counts\_per\_cycle}$$

where:

$\text{Offset}_{\text{degrees}}$ is the phase offset in degrees

$\text{Offset}_{\text{counts}}$ is the phase offset in encoder counts

counts_per_cycle is the # of counts per electrical cycle set using the SetPhaseCounts command

The phase offset value can also be changed any number of times while the motor is in motion, although only relatively small changes should be made to avoid sudden jumps in the motor motion.

The SetPhaseOffset and GetPhaseOffset commands can only be used when an index pulse from the encoder is connected. If no index pulse is used the phase offset angle cannot be adjusted or read back by the host.

Setting the phase offset value does not change the relative phasing of phase B and C to phase A These phases are still set at either 90 or 120 degree offsets from phase A (depending on the waveform chosen).

12.10 Encoder Pre-Scalar

Particularly when used with linear motors, the range in the value of the # of encoder counts per electrical cycle can vary widely. Typical rotary motors can have a value between 1 and 32,767. Linear brushless motors however can have values of 1,000,000 counts per cycle or higher because they often use high accuracy laser-based encoders.

To accommodate this large range the MC2300 series chips support a prescalar function which, for the purposes of commutation calculations, divides the incoming encoder counts by 64. With the prescalar enabled the max range for the number of encoder counts per electrical cycle is 2,097,088.

To enable the prescalar use the command SetPhasePrescale On. To disable the prescalar use the command SetPhasePrescale Off.

The prescalar function only affects the commutation of the chipset. It does not affect the position used during servo filtering or requested by the command GetActualPosition.

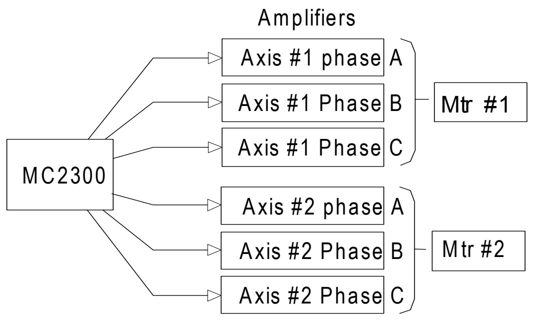
The prescalar function should not be enabled or disabled once the motor has been put in motion.

12.11 Motor Output Configuration

The MC2300 series of chipsets supports two motor output methods, PWM and DAC (up to 16 bit resolution).

Below is shown a typical amplifier configuration for a 3-phase brushless motor using the PWM output mode:

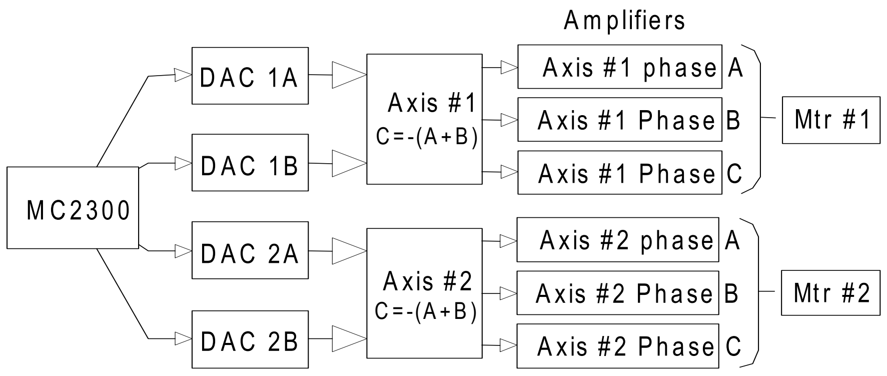
Brushless Motor (PWM Mode) Connection Scheme



In this configuration the chipset outputs 3 phased PWM magnitude signals per axis. These signals are then fed directly into 3 half-bridge type voltage amplifiers.

Below is shown a typical amplifier configuration for a 3-phase brushless motor using the DAC output mode:

Brushless Motor (DAC Mode) Connection Scheme

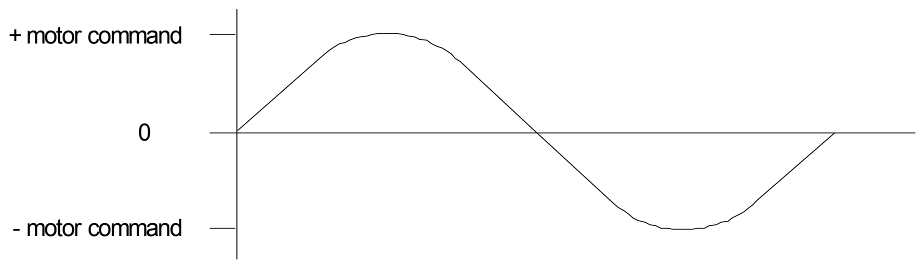


When using DAC output mode the digital word provided by the chipset must first be converted into a voltage using an external DAC. Two DAC channels are required per axis. To construct the third phase for a brushless motor (C phase) the sum of the A and B signals must be 'negated' using C = -(A+B).

This is usually accomplished with an Op-amp circuit. In addition, if current loop control is desired the three output signals are usually arranged so that the sum of the currents flowing through the windings of the motor are zero.

Motor Output Signal Interpretation

The following graph shows the desired output voltage waveform for a single phase.



The waveform is centered around a value of 0 volts. The magnitude of the generated waveform is proportional to either the output of the servo filter or the motor command register (depending on the commutation mode and motor on/off status).

For example if the chipset is connected to a DAC with output range of -10 Volts to +10 Volts and the chipset is set to open loop mode with a motor command value of 32,767 (which is the maximum allowed value) than as the motor rotates through a full electrical cycle, a sinusoidal waveform centered at 0 volts will be output with a minimum voltage of -10 and a maximum voltage of +10.

DAC Decoding

The digital values output by the chipset to the DAC encode the desired voltages as a 16-bit digital word. The minimum voltage is output as a digital word value of 0, a voltage of 0 Volts is output as a digital word of 32,768 (dec.), and the maximum positive voltage is output as a digital word value of 65,535.

To load each of the DACs, the DAC control pins in combination with the chipset's 16-bit data bus are used. To load a particular DAC, The DAC address (1 of 8) is output on the signals DAC16Addr0-3, the 16 bits of DAC data are output on pins Data0-16, I/OAddr0-3 and DACSlct are high, and I/OWrite is low.

For more information on the DAC signal timing & conditions, see the Pin Descriptions and timing diagrams section of the *Technical Specifications for Brushless Servo Motion Control.*

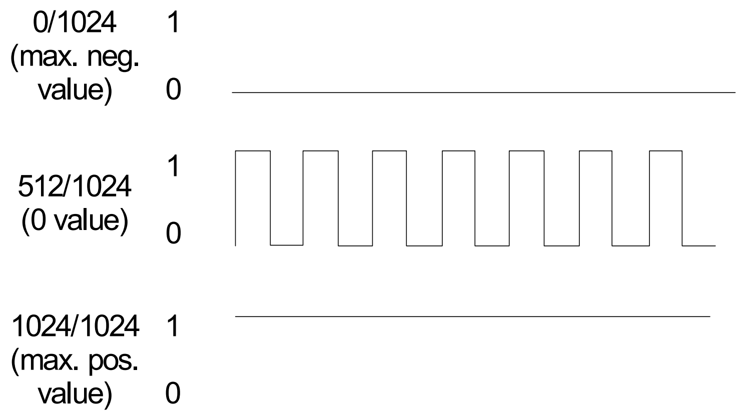
DACs with lower resolution than 16 bits can also be used. To connect to a DAC with less resolution, the high order bits of the 16-bit data word should be used. For example, to connect to an 8-bit DAC, bits Data7-Data15 should be used. The low order 8 bits are written to by the chipset, but ignored by the DAC circuitry.

PWM Decoding

The PWM output mode also outputs a sinusoidal desired voltage waveform for each phase, however the method by which these signals encode the voltage differ substantially from the DAC16 digital word.

The PWM output mode uses a single signal per output motor phase. This signal contains a pulse-width encoded representation of the desired voltage. In this encoding the duty cycle of the waveform determines the desired voltage. The PWM cycle has a frequency of 20 kHz, with a resolution of 10 bits, or 1/1,024.

The following chart shows the encoding:



An output pulse width of 0 parts per 1,024 represents the maximum negative voltage, an output pulse width of 512 per 1,024 (50 %) represents a voltage of 0, and a pulse width of 1,024 per 1,024 represents the maximum positive voltage.

This PWM scheme has been chosen to allow convenient interfacing to half bridge type amplifiers by connecting the PWM output to a level shifter circuit, and using this output to drive the high and low side drivers of the bridge.

13 Open Loop Stepper Control (MC2400, MC2500)

13.1 Overview

This chapter describes the open loop stepper control features of the MC2400 and MC2500 chipsets. Both of these chipsets contain circuitry for encoder feedback that can be used for detecting a stall in the motion of the attached motor in addition to output circuitry designed for stepper motor control.

The beginning of this chapter discusses the common stepper features of the MC2400 and MC2500, with the output mechanisms specific to each chipset being discussed in the second and third sections of the chapter.

13.1.1 Trajectory control units

For the Navigator stepping products, all units for trajectory control are steps for the MC2500 series and micro-steps for the MC2400 series. In the servo products (MC2100/MC2300/MC2800) all units are in encoder counts. The table below shows the commands and the appropriate units.

Command	MC2100/2300/2800	MC2400	MC2500
Set/GetPosition	counts	micro-steps	steps
Set/GetVelocity	counts/cycle	micro-steps/cycle	steps/cycle
Set/GetAcceleration	counts/cycle^2	micro-steps/cycle^2	steps/cycle^2
Set/GetDeceleration	counts/cycle^2	micro-steps/cycle^2	steps/cycle^2
Set/GetJerk	counts/cycle^3	micro-steps/cycle^3	steps/cycle^3
Set/GetStartVelocity	-	micro-steps/cycle	steps/cycle
GetCommandedPosition	counts	micro-steps	steps
GetCommandedVelocity	counts/cycle	micro-steps/cycle	steps/cycle
GetCommandedAcceleration	counts/cycle^2	micro-steps/cycle^2	steps/cycle^2
Set/GetPositionErrorLimit	counts	micro-steps	steps
GetPositionError	counts	micro-steps	steps

13.1.2 Encoder feedback

The Navigator stepping chipsets include support for an incremental encoder, or position data presented as a parallel word on the data bus. On power-up or a after a Reset instruction, the encoder source (GetEncoderSource) is set to *none*, making the encoder feedback optional. In this mode, the encoder position is ignored by the chipset. The current actual position is retrieved using the command GetActualPosition.

The SetActualPosition command can be used to set the current actual position to a programmed value. The default units of this command are encoder counts. To simplify program design and debugging, actual position units can be changed to steps/micro-steps. This is done using the command SetActualPositionUnits. The table below shows the commands that are affected by

Command	Position Units = counts	Position Units = steps
Set/GetActualPosition	counts	steps/micro-steps
AdjustActualPosition	counts	steps/micro-steps
GetCaptureValue	counts	steps/micro-steps

For further information on interfacing to encoders, refer to chapter 9.

13.1.3 Stall Detection

In addition to passively returning position to the host with the GetActualPosition command, the Navigator chipset can actively monitor the target and actual position and detect a motion error that results in a stall condition. Automatic stall detection allows the chipset to detect when the step motor has lost steps during a motion. This typically occurs when the motor encounters an obstruction, or otherwise exceeds its rated torque specification.

Automatic stall detection operates continuously once it is initiated. The current desired position (commanded position) is compared with the actual position (from the encoder) and if the difference between these two values exceeds a specified limit a stall condition is detected. The user programmed register SetPositionErrorLimit determines the threshold at which a motion error is generated.

To initiate automatic stall detection the host must specify the number of encoder counts per output step/micro-step. This is accomplished using the command SetEncoderToStepRatio. This command accepts two parameters, the first parameter is the number of encoder counts per motor rotation and the second parameter is the number of steps/micro-steps per motor rotation.

Parameter	Format	Word size	Range
Encoder counts per rev	16.0	16 bits	0 to 32,767
Steps/micro-steps per rev	16.0	16 bits	0 to 32,767

For example if a step motor with 1.8 degree full step size is used with an encoder which has 4,000 counts per motor rotation, the parameters used would be

SetEncoderToStepRatio 4000 200

where the number of steps per rotation is derived from 360/1.8.

If the same motor and encoder are used with the MC2400 and the number of micro-steps per full step is set to 64, then the parameters would be

SetEncoderToStepRatio 4000 12800

where the number of steps per rotation is derived from (360/1.8)*64.

In cases where the number of steps, micro-steps or encoder counts per rotation exceeds the allowed maximum of 32767, the parameters can be specified as a fractions of a rotation as long as the ratio is maintained accurately. So in the above example, the ratio could also be represented as

SetEncoderToStepRatio 2000 6400

indicating the ratio for half a revolution. Specifying the ratio for a fraction of a rotation is just as accurate as specifying it for a full rotation.

A typical sequence for enabling stall detection is shown below.

Command	Description
SetEncoderSource Incremental	Set the source for encoder position feedback
SetEncoderToStepRatio 4096 200	Set the ratio between the steps/micro-steps and the encoder counts per revolution
SetPositionErrorLimit	Set the desired error window in units of steps/micro-steps
ClearPositionError Update	Zero any existing position error
SetAutoStopMode On	Enables stopping the motor when a motion error is detected

At the moment a motion error occurs, several events occur simultaneously. The Motion Error bit of the event status word is set. If automatic stop on motion error is enabled the motor is set off, which has the effect of disabling the trajectory generator.

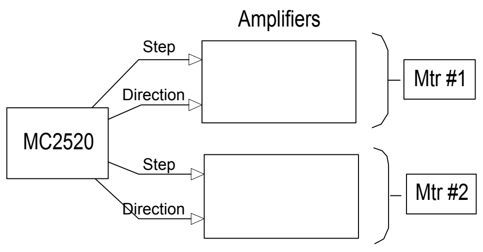
Recovering from a motion error is fully described in chapter 7.1.

13.2 Pulse & Direction Signal Generation (MC2500 only)

For each axis two signals are provided which determine the desired axis position at any given moment. These two signals are the pulse signal, and the direction signal.

The pulse signal output by the chipset consists of a precisely controlled series of individual pulses each of which represents a desired increment of movement. This signal is always output as a square wave pulse train (50 % duty cycle regardless of pulse rate). By default, a step, or pulse, is considered to have occurred when the pulse signal transitions from a high to a low output value. Inverting this logic is discussed later in this chapter. The direction signal is synchronized with the pulse signal at the moment each pulse transition occurs. The direction signal is encoded such that a high value indicates a positive direction pulse, and a low value indicates a negative direction pulse.

Step and Direction motor connection scheme



The MC2500 series of chipsets supports separate pulse rate modes using the command SetStepRange. The table below shows the values and resultant step ranges available using this command.

Command	Frequency range of output pulses
SetStepRange 1	0 to 4.98 M steps per second
SetStepRange 4	0 to 622.5 K steps per second
SetStepRange 6	0 to 155.625 K steps per second
SetStepRange 8	0 to 38.90625 K steps per second

The ranges above show the maximum and minimum ranges that can be generated by the chipset for the specified mode. So, for example, if the desired maximum step rate is 200 K steps per second, then the appropriate setting is SetStepRange 4.

For full-step and half-step applications, as well as pulse and direction applications which will have a maximum velocity of ~ 38 ksteps/sec, SetStepRange 8 should be used. For applications which require pulse rates higher than 48 ksteps/sec the higher speed ranges should be used.

A different step range can be set for each axis. To read back the current step range setting, use the GetStepRange command.

The pulse counter is designed such that a step occurs when the pulse signal transitions from high to low. Systems that use step motor amplifiers that interpret a pulse as a low to high transition should should use the command SetSignalSense to set the step logic to this mode. Refer to the *Programmer's Reference* for information on SetSignalSense.

13.2.1 Pulse Generation Control

The rate of pulse output is usually determined by the particular trajectory profile parameters being requested by the host processor. In addition to the trajectory profile however there is separate method of enabling and disabling pulse generation. This method is known as 'motor control' and provides an on/off pulse generator control mechanism.

The command to enable pulse output is SetMotorMode On and the command to disable pulse generation is SetMotorMode Off. SetMotorMode Off causes the trajectory generator to immediately discontinue further pulse generation until a SetMotorMode On command is given. As long as the motor is in the off state any further trajectory commands will have no effect until the motor is turned on.

The current motor status (on or off) can be read back using GetMotorMode.

If the motor is turned on by the host (SetMotorMode On command) the motor will stay at rest until a new trajectory move is loaded and initiated, if will not restart motion if a trajectory was previously programmed.

13.2.2 At Rest Indicator

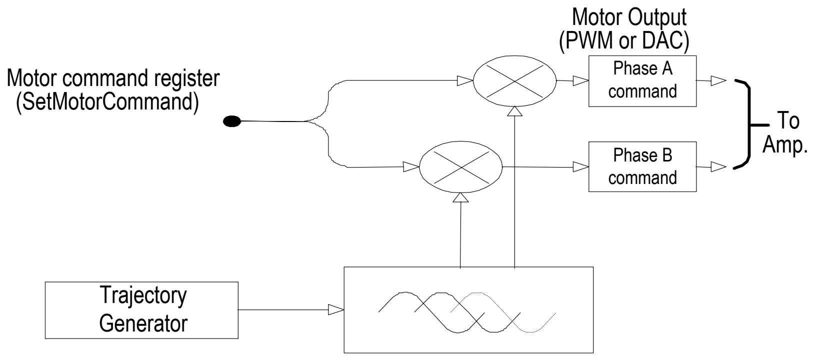
In addition to the standard pulse and direction output signals the MC2500 series chipsets provide an additional output for each axis known as the AtRest signal which indicates when the trajectory generator is in motion. This signal can be useful when interfacing with amplifiers that support a separate torque output level for the stepper during motion as when the motor is not moving (holding).

This feature is enabled and operational automatically at all times. It does not need to be enabled by the host processor.

13.3 Microstepping Waveform Generation (MC2400 only)

In addition to trajectory generation the MC2400 chipset provides direct internal generation of microstepping signals for 2-phase, as well as 3-phase stepper motors.

The following diagram shows an overview of the control flow of the microstepping scheme:



The microstepping portion of the chipset generates a sinusoidal waveform with a number of distinct output values per full step (one full step is one quarter of an electrical cycle). The number of microsteps per full step is set using the command SetPhaseCounts. The parameter used for this command represents the number of microsteps per electrical cycle (4 times the desired number of microsteps). So for example, to set 64 microsteps per full step, the command SetPhaseCounts 256 should be used. The maximum number of microsteps that can generated per full step is 256, giving a maximum parameter for this command of 1024.

The output frequency of the microstepping signals are controlled by the trajectory generator. The amplitude of the microstepping signals are controlled using a register that can be set by the host processor known as the motor command register. Adjustment of this register by the host allows different motor power levels during (for example) motion, and at rest.

Two microstepping waveforms are provided, one appropriate for traditional 2-phase stepper motors with 90 deg. of separation between phases, and one appropriate for 3-phase stepper motors and AC Induction motors with 120 deg. separation between phases. For more information on AC Induction Motor Control see the section entitled AC Induction Motor Control.

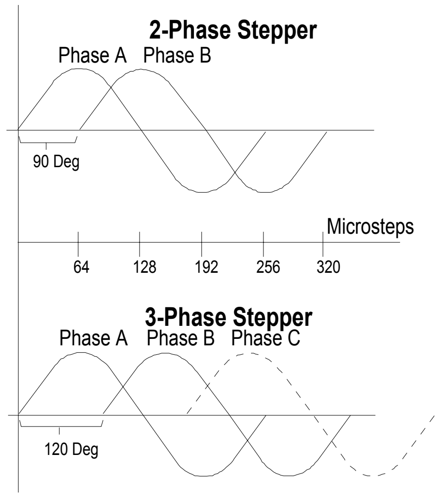
13.3.1 Microstepping Waveforms

To specify 2-phase motor waveforms use the command SetNumberPhases 2, and to specify 3-phase motor waveforms use the command SetNumberPhases 3.

Waveform	Motor output mode	Number of output signals & name
2-phase	PWMSign/Mag	2 (A, B)
2-phase	DAC	2 (A, B)
3-phase	PWM50/50	3 (A, B, C)
3-phase	DAC	2 (A, B)

For specific pin assignments of the PWM and DAC motor output signals see the Navigator Technical Specifications manual appropriate for your chipset.

The diagram below shows the phase A, B signals for a 2-phase stepper motor, and the phase A, B signals for a 3-phase stepper motor or AC Induction motor.



For 3-phase stepper motors or AC Induction motors, the phase C waveform must be constructed externally using the expression C = -(A+B). Typically this is performed by the motor amplifier. See the following section of this manual entitled "Motor Output" for more information.

13.3.2 Motor Command Control

The MC2400 provides the ability to set the motor command (power output) level of the stepper motor. This is often useful to optimize the motor torque, power consumption, and heat generation of the motor while it is at rest, or in various states of motion.

The motor output level is controlled by the motor command register. This register can be set using SetMotorCommand. A value between 0 and 32767 is set, representing an amplitude of zero to 100 percent. Since SetMotorCommand is double buffered, it requires an Update or a breakpoint to occur before it takes effect. This feature can be used to advantage when it is desired that the motor power changes be synchronized with other profile changes such as at the start or the end of a move.

Changing the power level does not affect the microstepping output phasing or the frequency of the output waveform, it simply adjusts the magnitude of the waveform.

13.3.3 AC Induction Motor Control

The MC2400 chipset can be used as the basis of a variable speed 3-phase AC Induction motor controller. In this mode the chipset is set for a 3-phase waveform, and is operated as if it were a stepper motor. The position of the motor is not precisely maintained, however the velocity of the AC Induction motor can typically be controlled to within 10 - 20 percent. Such a controller can be used for spindles, and other motors where velocity control, not positioning is required.

When running an AC Induction motor using variable speed control care should be taken that the output drive signal should never have a frequency of 0. Even if the motor is not rotating the drive frequency should have at least some rotational frequency. This is because a relative difference in the frequency of the drive signals and the motor rotor (called the slip frequency) is required to avoid magnetic field saturation at rest, a potentially damaging condition.

Using the MC2400 up to four, two or one AC Induction motor can be controlled. The output drive configuration is the same as for 3-phase steppers shown in the 'Motor Output Configuration' section.

The MC2400 chipset does not provide 'Flux Vector Control' of AC Induction Motors, only variable speed control. Therefore the MC2400 should not be used in AC Induction applications involving precision positioning.

Command Summary

The following table summarizes the commands that are used in conjunction with microstepping signal generation:

Command	Function
SetOutputMode	Sets the output to either PWM or DAC
SetNumberPhases	Sets the number of motor phases
SetMotorCommand	Sets the amplitude of the output waveform. This is a buffer command and requires an Update
SetPhaseCounts	Sets the number of microsteps per electrical cycle of the motor

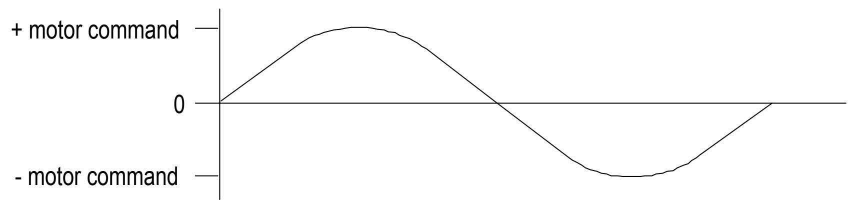
Before the chipset will generate any motor output, the motor command value must be set and the appropriate output mode and number of phases must be set.

13.4 DAC and PWM Motor Output (MC2400 only)

The MC2400 series of chipsets support two motor output methods, PWM and DAC. The motor output method is host-selectable. The method can be selected individually for each axis. The host command to select the output mode is SetMotorCommand with the parameter specifying the output method. A value of zero sets the output type to DAC and a value of one sets the output to PWM sign/magnitude.

13.4.1 Motor Output Signal Interpretation

The diagram below shows typical waveforms for a single output phase of the MC2400 chipset. Each phase has a similar waveform, although the phase of the B channel output is shifted relative to the A channel output by 90 or 120 degrees (depending on the waveform selected).

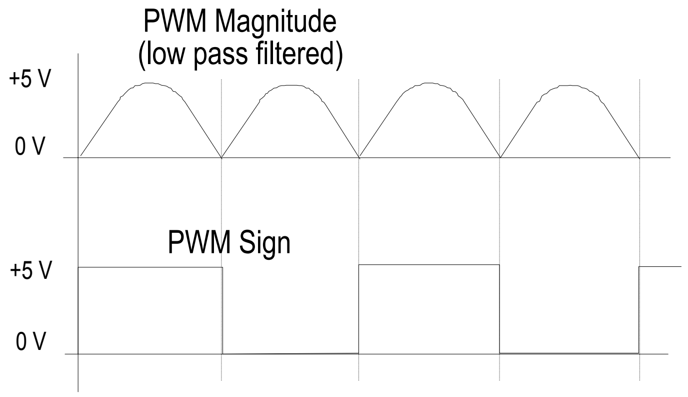


The waveform is centered around an output value of 0. The magnitude of the overall generated waveform is controlled by the motor command register (SetMotorCommand).

For example if the chipset is connected to a DAC with an output range of -10 Volts to + 10 Volts and the chipset is set to a motor command value of 32,767 (which is the maximum allowed value) than as the motor rotates through a full electrical cycle, a sinusoidal waveform centered at 0 volts will be output with a minimum voltage of - 10, and a maximum voltage of +10.

13.4.2 PWM Decoding

The PWM output mode also outputs a sinusoidal desired voltage waveform for each phase, however the method by which these signals encode the voltage differ substantially from the DAC digital word. The PWM mode uses a magnitude signal and a sign signal. The magnitude signal encodes the absolute value of the output sinusoid and the sign signal encodes the polarity of the output, positive or negative. The following diagram shows the magnitude and sign signals for a single output phase.



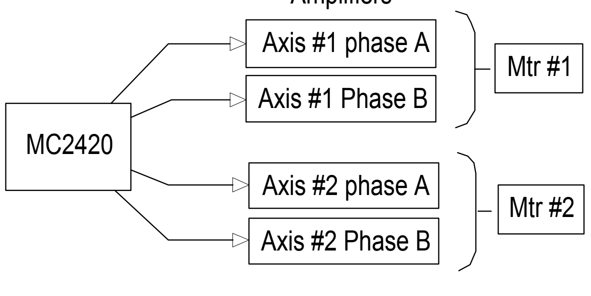
In this diagram the PWM magnitude signal has been filtered to convert it from a digital variable duty cycle waveform to an analog signal.

Before filtering this signal contains a pulse-width encoded representation of the 'analog' desired voltage. In this encoding the duty cycle of the waveform determines the desired voltage. The PWM cycle has a frequency of 78.124 KHz, with a resolution of 8 bits, or 1/256.

13.4.3 Motor Drive Configurations

Shown below is a typical amplifier configuration for a 2-phase stepper motor using either the PWM or DAC output mode.

2-Phase Motor Output Connection Scheme



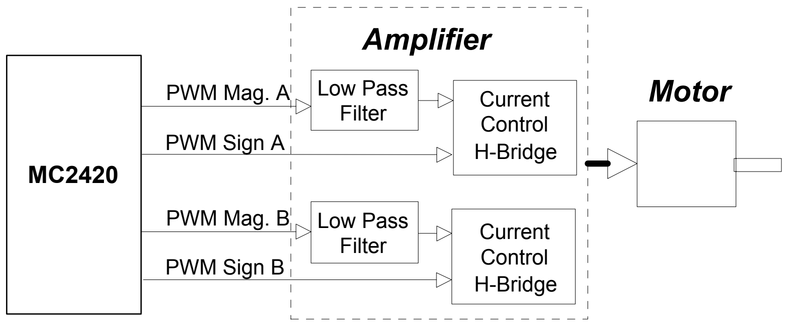
Using the DAC output mode the digital motor output word for each phase is typically converted into a DC signal with a value between -10 to +10 volts. This signal can then be input into an off-the-shelf DC-Servo type amplifier (one amplifier for each phase) or into any other linear or switching amplifier that performs current control and provides a bipolar, two-lead output.

In this scheme each amplifier drives one phase of the stepper motor, with the chipset generating the required sinusoidal waveforms in each phase to perform smooth, accurate motion.

If the chipset's PWM output mode is used the PWM magnitude and sign signals are typically connected to an H-bridge-type device. For maximum performance, current control should be performed by the amplifier. This minimizes the coil current distortion due to inductance and back-EMF. Although there are several methods that can be used to achieve current control with the PWM output mode, a common method is to pass the PWM magnitude signal through a low pass filter, thereby creating an analog reference signal which can be directly compared with the current through the coil.

Several single-chip amplifiers are available which are compatible with these input signals. These amplifiers require an analog reference input (low-passed PMWMag signal from chipset) as well as a sign bit (PWMSign signal from chipset). The amplifier in-turn performs current control typically, using a fixed-off time PWM drive scheme. See the *Navigator MC2400 Technical Specifications* manual for an example of such a circuit.

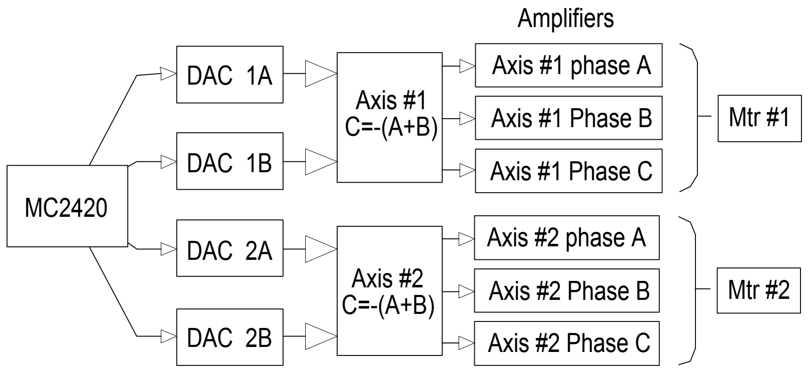
The diagram below shows this amplifier scheme:



Relative to the DAC output method the PWM output mode when used with this amplifier scheme has the advantage of high performance with a minimum of external parts.

Below is shown a typical amplifier configuration using the MC2420 in DAC mode for a 3-phase stepper or for an AC Induction motor with 3 phases.

3-Phase DAC Connection Scheme



When using DAC output mode the digital word provided by the chipset must first be converted into a voltage using an external DAC. Two DAC channels are required per axis. The third phase is constructed externally using the expression C = -(A+B). This is usually accomplished with an Op-amp circuit.

# # #

For additional information, or for technical assistance, please contact PMD at (781) 674-9860. You can also email your request to mailto:apps@pmdcorp.com. Visit our website at http://www.pmdcorp.com/.