

NavigatorTM Motion Processor

User's Guide



P M D

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

Revision 1.0, June, 1999

Copyright 1998, 1999 by Performance Motion Devices, Inc.

Navigator and **C-Motion** are trademarks of Performance Motion Devices, Inc.

NOTICE

This document contains proprietary and confidential information of Performance Motion Devices, Inc., (PMD) and is protected by federal copyright law. The contents of this document may not be disclosed to third parties, translated, copied, or duplicated in any form, in whole or in part, without the express written permission of PMD.

The information contained in this document is subject to change without notice. No part of this document may be reproduced or transmitted in any form, by any means, electronic or mechanical, for any purpose, without the express written permission of PMD.

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

Four 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).

Navigator Motion Processor Developer's Kit Manual (DK2000M)

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

Table of Contents

Warranty	iii
Safety Notice.....	iii
Disclaimer.....	iii
Related Documents	iv
Table of Contents.....	v
The Navigator Family.....	7
1 System Overview.....	9
2 Trajectory Generation.....	11
2.1 Trajectories, profiles, and parameters	11
2.2 Trapezoidal point-to-point profile.....	11
2.3 S-curve point-to-point profile	13
2.4 Velocity-contouring profile	15
2.5 Electronic-gear profile	17
2.6 The SetStop command.....	18
2.7 Motor Mode.....	19
2.8 Setting the cycle time.....	19
3 The Servo Loop.....	21
3.1 Overview.....	21
3.1.1 PID loop algorithm	21
3.1.2 Motor bias.....	22
3.1.3 Output limit.....	22
3.2 Closed-loop and open-loop control modes	22
4 Parameter update and breakpoints.....	24
4.1 Parameter buffering	24
4.1.1 Updates	24
4.2 Breakpoints.....	25
4.2.1 Defining a breakpoint, Overview.....	25
4.2.2 Breakpoint triggers	26
4.2.3 Threshold-triggered breakpoints	26
4.2.4 Level-triggered breakpoints	26
4.2.5 Breakpoint actions	27
4.2.6 Breakpoint Examples	27
5 Status Registers.....	29
5.1 Overview.....	29
5.1.1 Event Status register	29
5.1.2 Instruction error	30
5.1.3 Activity Status register.....	31
5.1.4 Signal Status	32
5.1.5 Signal Sense Mask	32

6	Monitoring Motion Performance	33
6.1	Motion Error	33
6.1.1	Automatic Stop On Motion Error	33
6.2	Tracking window	34
6.3	Motion Complete Indicator.....	34
6.4	In-motion indicator	35
6.5	Settled indicator	35
6.6	Data Trace.....	37
6.6.1	The trace buffer.....	38
6.6.2	The trace period	38
6.6.3	Trace variables.....	38
6.6.4	Trace modes	40
6.6.5	Trace start/stop conditions	40
6.6.6	Downloading Trace Data	41
6.6.7	Running Traces	41
6.7	Host Interrupts	42
7	Hardware Signals	44
7.1	Travel-limit switches	44
7.2	The <i>AxisOut</i> pin	45
7.3	The <i>AxisIn</i> pin.....	46
8	Motor Interfacing	47
8.1	Incremental Encoder Input.....	47
8.1.1	Actual Position Register	47
8.1.2	Digital Filtering.....	47
8.1.3	High speed position capture	48
8.2	Parallel-word position input.....	48
8.2.1	Multi-turn systems	49
8.2.2	Parallel-word device interfacing	49
8.3	Peripheral Device I/O	50
8.3.1	Peripheral Device Read.....	50
8.3.2	Peripheral Device Write.....	50
8.4	Motor Command Output.....	51
9	Host Communication.....	54
9.1	Primary and Diagnostic Ports	54
9.1.1	Diagnostic port functions.....	54
9.2	Parallel Communication Port.....	55
9.2.1	Control signals	56
9.2.2	Parallel port I/O operations	56
9.2.3	The <i>status read</i> operation.....	57
9.3	Parallel Host I/O Commands	57
9.4	Serial Port	60
9.4.1	Configuration	60
9.4.2	Control signals	61
9.4.3	Command format	62
10	Using External Memory	65
10.1	Memory configuration	65
10.1.1	Memory page pointer.....	65
10.1.2	External memory signal decoding.....	65
10.1.3	External memory read.....	66
10.1.4	External memory write	66
10.1.5	External memory buffers	66
10.1.6	External memory commands	67

The Navigator Family

	MC2100 Series	MC2300 Series	MC2400 Series	MC2500 Series
# of axes	4, 2, or 1	4, 2 or 1	4, 2 or 1	4, 2, or 1
Motors Supported	Brushed servo	Brushless servo	Stepping	Stepping
Encoder Format	Incremental and parallel-word	Incremental and parallel-word	Incremental and parallel-word	Incremental and parallel-word
Output Format	Brushed servo (single phase)	Commutated (6-step or sinusoidal)	Microstepping	Pulse and direction
Communication options	Parallel and serial	parallel and serial	parallel and serial	parallel and serial
Diagnostic port	Yes	Yes	Yes	Yes
S-curve profiling	Yes	Yes	Yes	Yes
Electronic gearing	Yes	Yes	Yes	Yes
On-the-fly changes	Yes	Yes	Yes	Yes
Limit switches	Yes	Yes	Yes	Yes
Programmable bit output	Yes	Yes	Yes	Yes
Software-invertable signals	Yes	Yes	Yes	Yes
PID servo control	Yes	Yes	-	-
feedforward (accel & vel)	Yes	Yes	-	-
derivative sampling time	Yes	Yes	-	-
Data trace/diagnostics	Yes	Yes	Yes	Yes
PWM output	Yes	Yes	Yes	-
Motion error detection	Yes	Yes	Yes (with encoder)	Yes (with encoder)
Axis settled indicator	Yes	Yes	Yes (with encoder)	Yes (with encoder)
DAC-compatible output	Yes	Yes	Yes	-
Pulse & direction output	-	-	-	Yes
Index & Home	Yes	Yes	Yes	Yes
Position capture	Yes	Yes	Yes	Yes
Analog input	Yes	Yes	Yes	Yes
User-defined I/O	Yes	Yes	Yes	Yes
External RAM support	Yes	Yes	Yes	Yes
Chipset p/n's	MC2140 (four axes) MC2120 (two axes) MC2110 (one axis)	MC2340 (four axes) MC2320 (two axes) MC2310 (one axis)	MC2440 (four axes) MC2420 (two axes) MC2410 (one axis)	MC2540 (four axes) MC2520 (two axes) MC2510 (one axis)
Developer's Kit p/n's:	DK2100	DK2300	DK2400	DK2500

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), and MC2500 Series (stepping) chipsets.

Each of these devices is a chip-based motion processor. They provide trajectory generation and related motion control functions. Depending on the type of motor controlled they provide servo loop closure, on-board commutation, 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 motion control problems.

Each of these chips utilize 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 such as a multiply instruction 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.

MC2100 Series (MC2140, MC2120, MC2110) – These chipsets output 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.

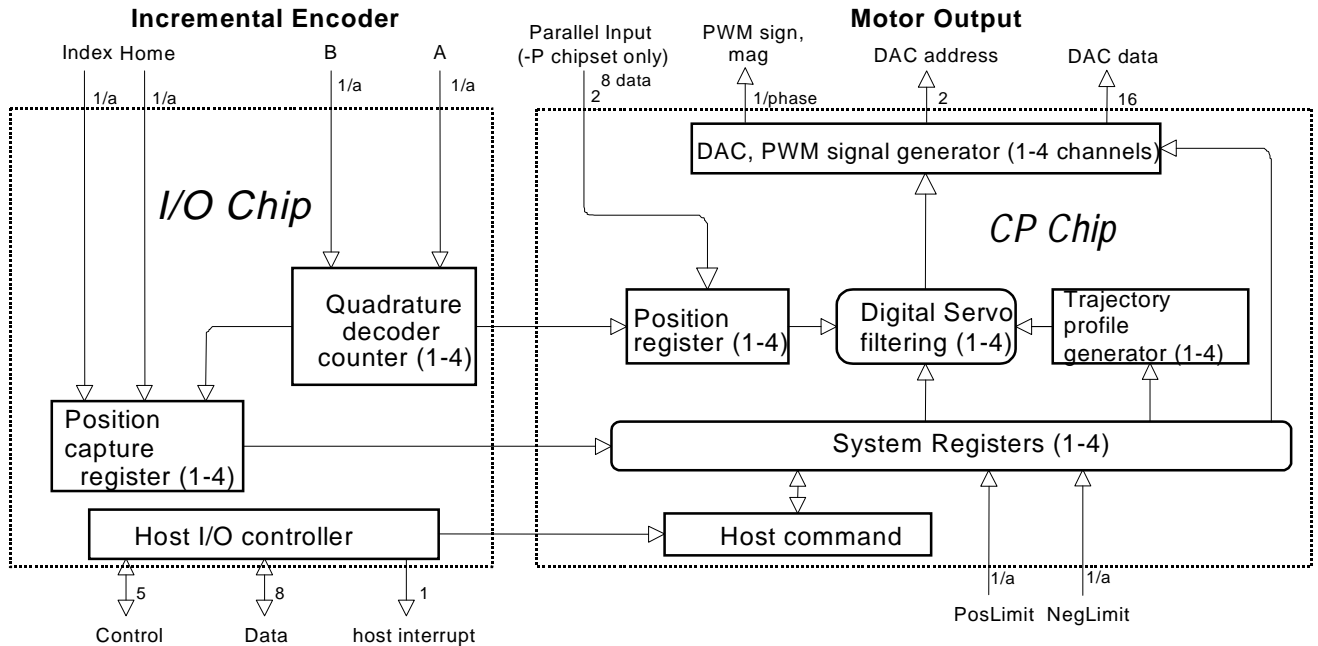
MC2300 Series (MC2340, MC2320, MC2310) – These output sinusoidally commutated motor signals appropriate for driving brushless motors. Depending on the motor type, the output is a two-phase or three-phase signal in either PWM or DAC-compatible format.

MC2400 Series (MC2440, MC2420, MC2410) – These chipsets provide 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.

1 System Overview

Internal Block Diagram



The above figure shows an internal 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 trajectory generator 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 or MC2300 Series), and some other chipset functions.

For the servo control chipsets (MC2100 and MC2300) 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 chipset (brushless servo) 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 MC2100 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 microprocessor-style 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, parameters, or conditions to be monitored and compared against a setpoint condition value. 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 into 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 switch - 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 and MC2300 Series only. If this is the case then these sections will be clearly marked as such.

2 Trajectory Generation

2.1 Trajectories, profiles, and parameters

The trajectory generator performs calculations to determine the instantaneous position, velocity and acceleration of the desired chipset at any given moment in time. These values are called the *commanded* values. During a profile motion some or all of these parameters will change continuously. Once the profile 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 Manual 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
- 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.

2.2 Trapezoidal point-to-point profile

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

Profile Parameter	word size	Range
Position	32 bits	- 2,147,483,648 to 2,147,483,647 counts
Velocity	32 bits	0 to 16,383+65,535/65,536 counts/cycle
Acceleration	32 bits	0 to 16,383+65,535/65,536 counts/cycle ²
Deceleration	32 bits	0 to 16,383+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 2-1a): 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. 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 2-1b).

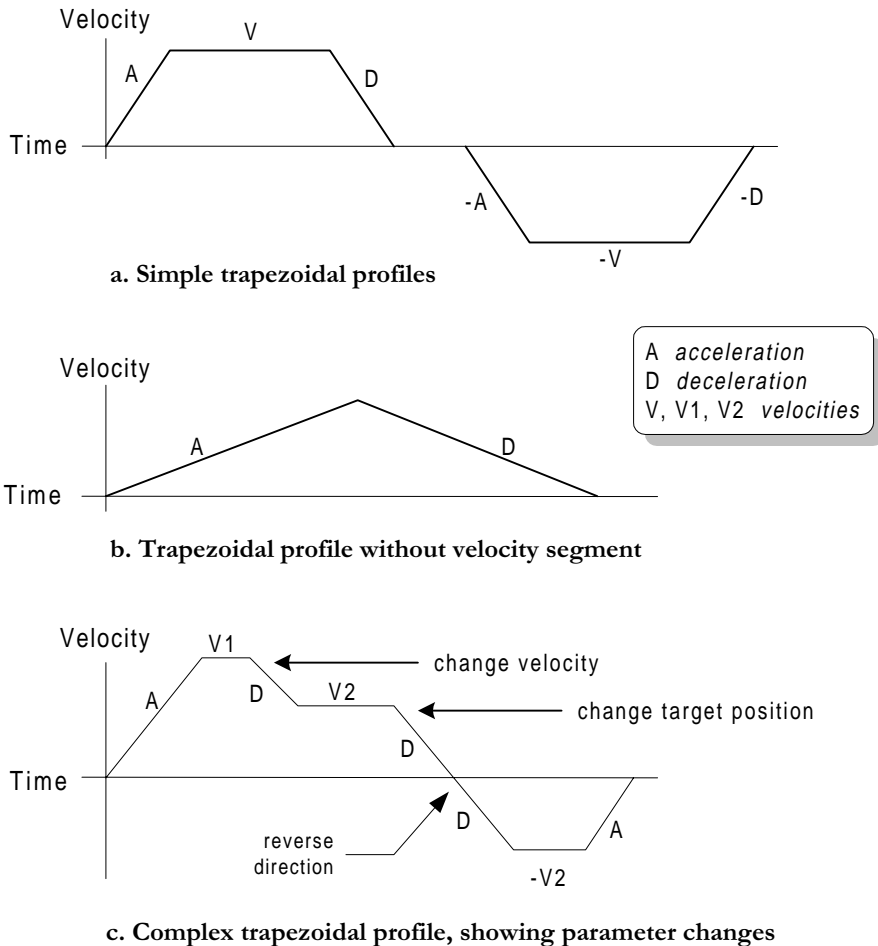


Figure 2-1. Trapezoidal point-to-point profiles

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 for acceleration in that direction, and deceleration will be used for accelerations in the opposite direction. 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.

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 2-1c.

If a deceleration value of 0 (zero) is programmed (or no value is programmed leaving the chipset's default value of zero) then the deceleration value used will be the acceleration value.

2.3 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	Range
<i>Position</i>	32 bits	- 2,147,483,648 to 2,147,483,647 counts
<i>Velocity</i>	32 bits	0 to 16,383+65,535/65,536 counts/cycle
<i>Acceleration</i>	32 bits	0 to 16,383+65,535/65,536 counts/cycle ²
<i>Deceleration</i>	32 bits	0 to 16,383+65,535/65,536 counts/cycle ²
<i>Jerk</i>	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

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.

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

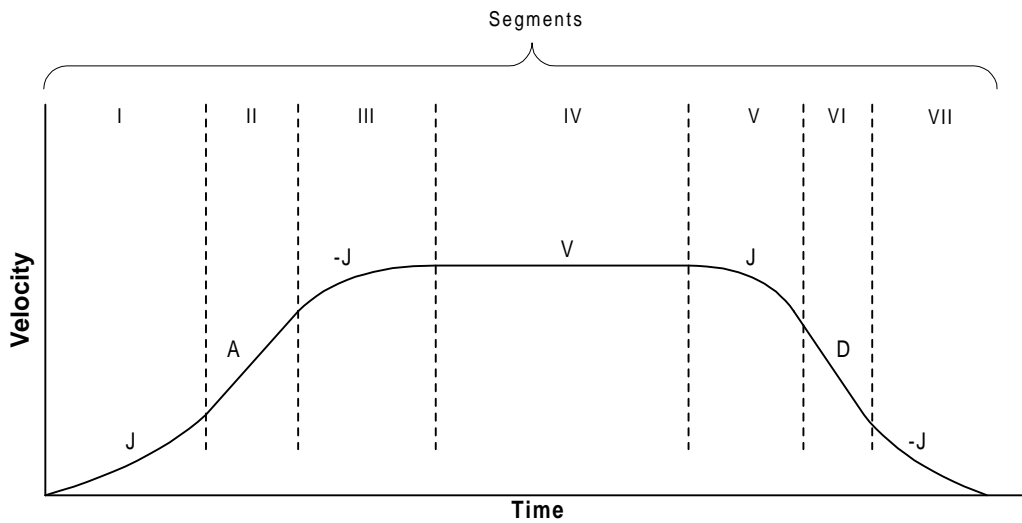


Figure 2-2. S-curve profile

Figure 2-2 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 in motion 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 (A), 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 2-2. 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 figure2-4

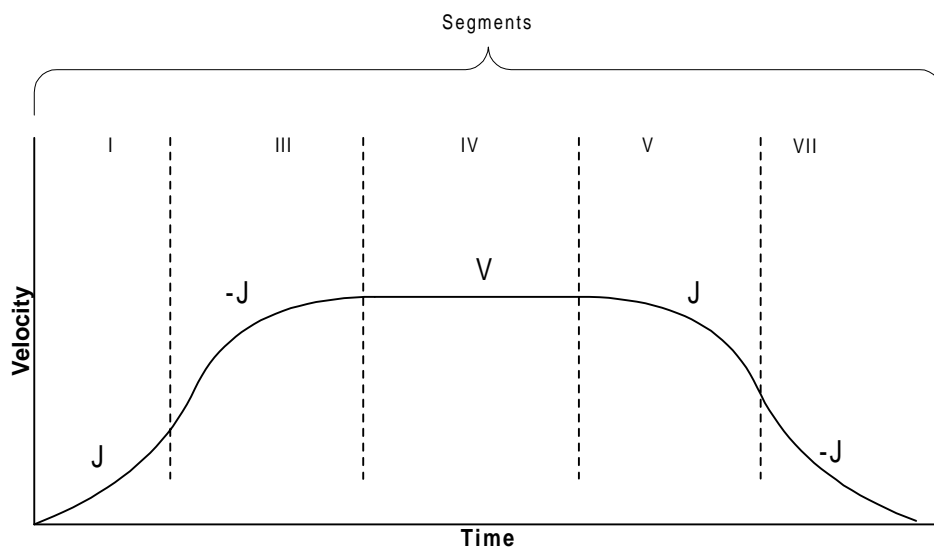


Figure 2-3. S-curve that doesn't reach maximum acceleration

Similarly, if the position is specified such that velocity is not reached, there will be no Segment IV, as shown in figure 2-4. (There may also be no Segment II or Segment VI, depending on where the profile is "truncated".)

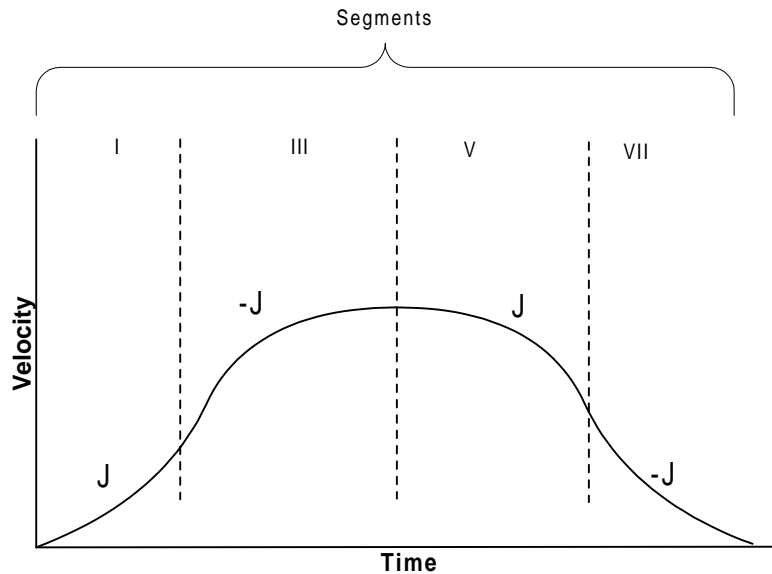


Figure 2-4. S-curve with no maximum-velocity segment

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. Likewise, the 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.

2.4 Velocity-contouring profile

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

Profile Parameter	Word size	Range
Velocity	32 bits	-16,384 to 16,383+65,535/65,536 counts/cycle
Acceleration	32 bits	0 to 16,383+65,535/65,536 counts/cycle ²
Deceleration	32 bits	0 to 16,383+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.

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.

A simple velocity-contouring profile looks just like a simple trapezoidal point-to-point profile, as shown in Figure 2-1.

Figure 2-5 illustrates a more complicated profile, in which the both the velocity and the direction of motion change twice:

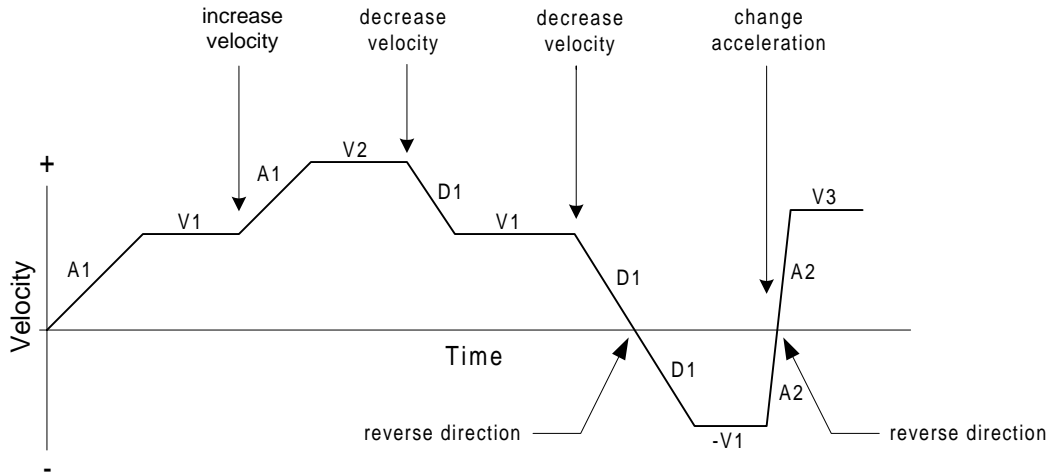


Figure 2-5. Velocity-contouring profile

NOTES

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.

2.5 Electronic-gear profile

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

Profile Parameter	word size	Range
<i>Gear ratio</i>	32 bits	-16,384 to 16,383+65,535/65,536 counts/cycle
<i>Master axis #</i>	4 bits	0 - 3
<i>Master source</i>	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 **SetRatio**, and **SetGearMaster** load these respective values. The commands **GetRatio** and **GetGearMaster** retrieve the programmed values.

In this profile, the host specifies three parameters. The first is the 'master' axis # 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 the 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.

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 is 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.

Figure 2-6 shows the arrangement of encoders and motor drives in a typical electronic gearing application.

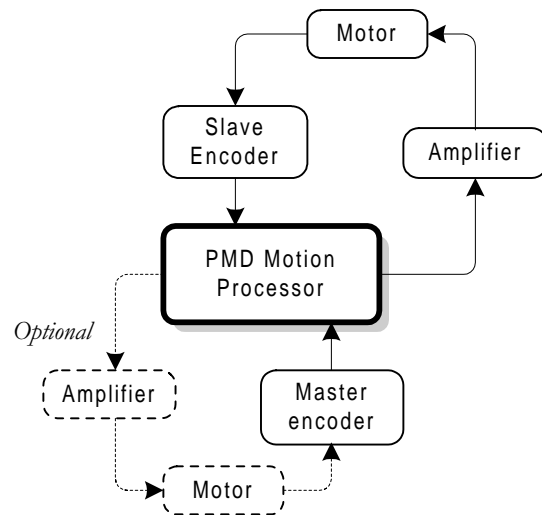


Figure 2-6. MC2100 set up for electronic-gearing profile

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 2-6 illustrate this arrangement. Such a configuration can be used to perform useful functions such as linear interpolation of two axes.

NOTES

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.

2.6 The SetStop 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 **SetStop** is used. To retrieve the current stop mode the command **GetStop** is used.

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.

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

CAUTION

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.

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

In addition, for the servo chipsets (MC2100 and MC2300), 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 6.1.

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

2.8 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:

Chipset	Cycle time per enabled axis
MC2100	100 μ sec
MC2300	150 μ sec
MC2400	100 μ sec
MC2500	N/A

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 an MC2100 with four axes enabled, the minimum loop time would be $4 \times 100 = 400 \mu$ sec, 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 and MC2300). It does not however determine the commutation rate of the MC2300 Series.

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 a 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 μ sec 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.

3 The Servo Loop

3.1 Overview

For the servo-based chipsets (MC2100 and MC2300) 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.

3.1.1 PID loop algorithm

The servo filter used with the MC2100 and MC2300 chipsets is a proportional-integral-derivative (PID) algorithm, with velocity and acceleration feedforward 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.

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

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

where

- I_{lim} is the Integration Limit
- 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 16 bits (0 to 32767)
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 (-32767 to 65,535)
	Motor command limit	unsigned 16 bits (0 to 32767)

The structure of the digital filter is shown in figure 3-2.

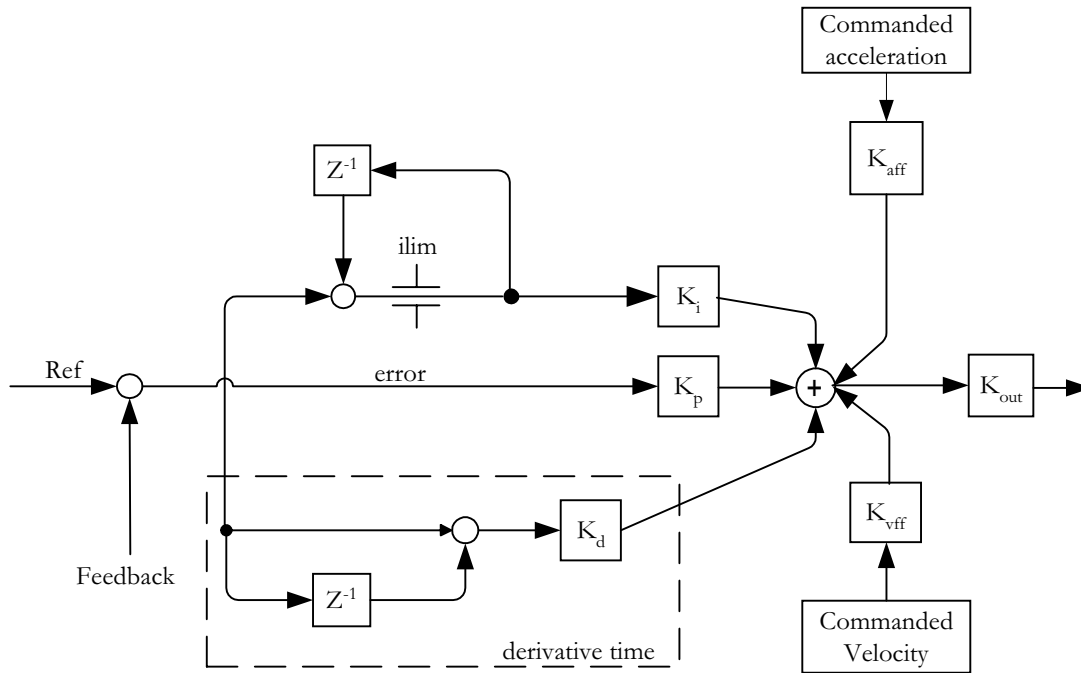


Figure 3-2. Digital Servo Filter

3.1.2 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 axis, bias**. It can be read back using the command **GetMotorBias axis**.

3.1.3 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 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).

3.2 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 and MC2300) 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 3-1 shows the control flow for open and closed loop operation.

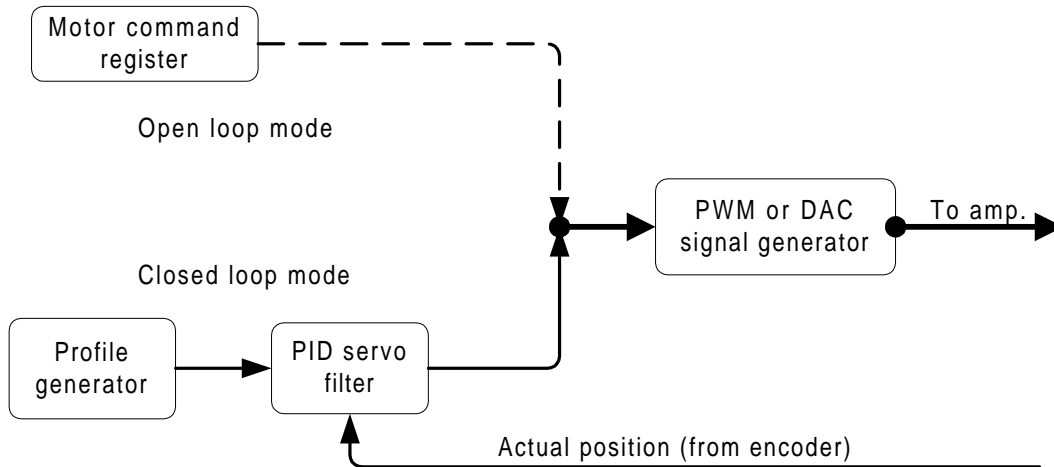


Figure 3-1. Motor control paths, closed- and open-loop modes

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

NOTE: Limit switches do not function in open-loop mode.

2.1.1 *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.

NOTE

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.

4 Parameter update and breakpoints

4.1 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 and MC2300 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 destination position in but will not become active (take affect) 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.

4.1.1 Updates

There are three different ways that an Update can occur. The 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 which consists of a bit mask, with 1 bit assigned to each axis. Executing this command has the same affect 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 facility 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 4.2

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. Conversely, before the Update occurs, loading the buffered registers or executing 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

<i>Trajectory</i>	<i>Servo & Error Tracking</i>	<i>Miscellaneous</i>
SetProfileMode	ClearPositionError	SetMotorCommand
SetAcceleration	SetIntegrationLimit	
SetJerk	SetKaff	
SetVelocity	SetKd	
SetPosition	SetKi	
SetDeceleration	SetKp	
SetGearRatio	SetKvff	
	SetKaff	

4.2 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 whatever.

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

4.2.1 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.

4.2.2 Breakpoint triggers

The Navigator motion processors support the following breakpoint trigger conditions:

Trigger Condition	Level or Threshold	Description
PositiveCommandedPosition	threshold	Is satisfied when the current commanded position is equal to or greater than the programmed compare value
NegativeCommandedPosition	threshold	Is satisfied when the current commanded position is equal to or less than the programmed compare value
PositiveActualPosition	threshold	Is satisfied when the current actual position is equal to or greater than the programmed compare value
NegativeActualPosition	threshold	Is satisfied when the current actual position is equal to or greater 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.

4.2.3 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 **SetComparisonValue** would be 1,000,000, and the trigger selected would be **PositiveCommandedPosition**.

4.2.4 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 than 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 cause 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.

4.2.5 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 <i>axis</i>
Abrupt Stop	SetStop <i>axis</i> , UbruptStop Update <i>axis</i>
SmoothStop	SetStop <i>axis</i> , SmoothStop Update <i>axis</i>
MotorOff	SetMotorMode <i>axis</i> , Off Update <i>axis</i>

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.

4.2.6 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, 100,000 // Load 100,000 into the comparison register
// for breakpoint 1
SetBreakpoint Axis1, 1, Axis1, Update, PositiveActualPosition
// Specify 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. For example 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 5,000 but do not send
// an Update
SetBreakpointValue Axis1, 2, 100,000 // 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. 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 affect once the breakpoint occurs) before the primary move is Updated.

5 Status Registers

5.1 Overview

The Navigator Motion Processor can monitor almost every aspect of an axis's motion. There are numerous numerical registers that can be queried to determine the current state of 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 7.2) mechanism, which allows any bit within these three registers to be output as a hardware signal.

5.1.1 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 wrap.	Set when the actual motor position exceeds 7FFFFFFh (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	<i>Reserved</i>	<i>May contain 1 or 0</i>
11	Commutation Error	Set when a commutation error occurs (MC2300 series only)
12,13	<i>Reserved</i>	<i>May contain 1 or 0</i>
14	Breakpoint 2	Set when breakpoint #2 is triggered.
15	<i>Reserved</i>	<i>May contain 0 or 1</i>

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.

5.1.2 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 however. 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)
SetProfileMode (axis2, Trapezoidal)
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.)

5.1.3 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: <table border="0" style="margin-left: 20px;"> <tr> <td>bit 5</td> <td>bit 4</td> <td>bit 3</td> <td>Profile Mode</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> <td>trapezoidal</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>velocity contouring</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>s-curve</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>electronic gear</td> </tr> </table>	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
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 6.5																				
7	<i>Reserved</i>	<i>May contain 0 or 1</i>																				
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 and MC2300) it means the chipset is in closed loop mode and the servo loop is operating. When the motor is off this means trajectory operations can not be performed and for the servo chipsets (MC2100 and MC2300) 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.

5.1.4 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).

5.1.5 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.

6 Monitoring Motion Performance

6.1 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 following list describes these events:

- 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 the disabling the trajectory generator (for all chipsets) and for the servo chipsets (MC2100 and MC2300) has the additional affect 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 following sequence should be performed:

1. Determine cause of motion error and correct problem (this may require human intervention).
2. Issue a **SetMotorMode axis, On** command followed by an **Update** 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.

6.1.1 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 and MC2300) 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 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, so that 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 the Servo Filter sections of this document. The motor bias is a fixed, open-loop command to the motor, which is normally 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.

This means that with a properly set motor bias, when 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.

6.2 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 6-1.

6.3 Motion Complete Indicator

In many cases it is useful to have the chipset signal that a give motion is completed. 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-based; the motion complete indicator is set based on the profile generator position only. The other method is actual-based, 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 commanded velocity and acceleration values 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 (`SetStop` command), a change of velocity to zero, or when a limit switch event occurs.

When set to actual-based 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-based 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.

6.4 In-motion indicator

The chipset can indicate whether or not the axis is moving. This function is available in 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.

6.5 Settled indicator

The chipset can also continuously indicate whether or not the axis is '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-based. The differences are that the settled indicator continuously indicates its status (can not be set or cleared) and also that it indicates regardless of whether or not the motion complete mode is set to actual based.

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-based. Correspondingly the same commands are used to set and read these values: **SetSettleWindow**, **GetSettleWindow**, **SetSettleTime**, **GetSettleTime**.

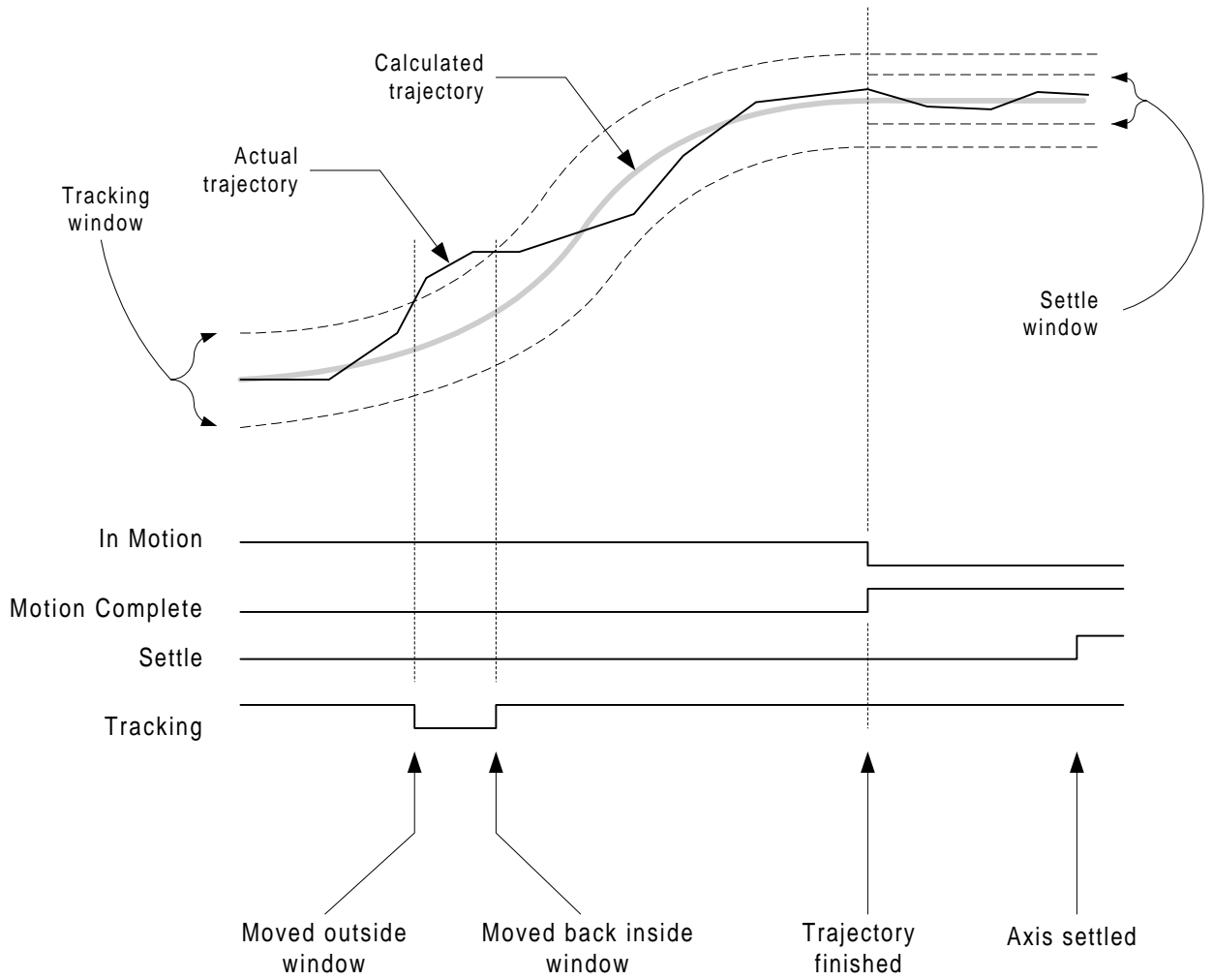


Figure 6-1. The tracking window

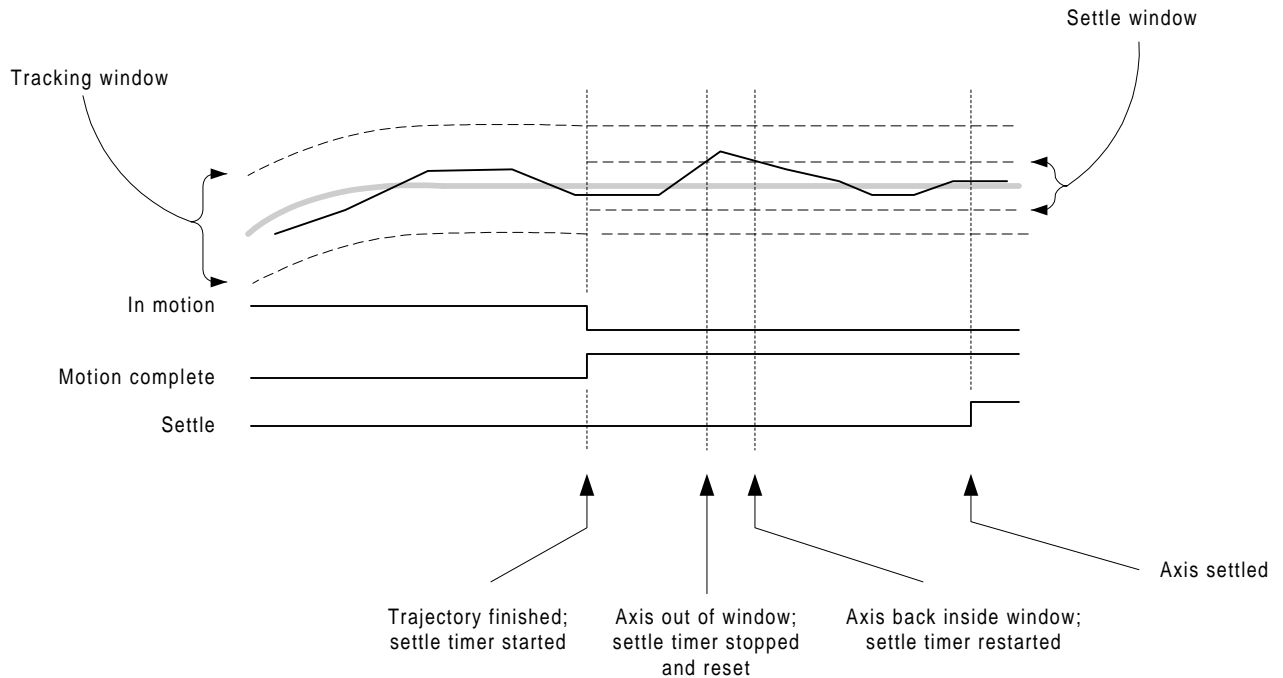


Figure 6-2. The settle window

6.6 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 data trace.

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.

6.6.1 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 10.1.5 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.

6.6.2 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, the command **GetTracePeriod** retrieves it.

6.6.3 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 trace *parameter* 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.
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 Torque	Commanded motor torque (output of the servo filter).
8	Chipset Time	The chipset time (units of servo cycles).
9	Capture Value	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	Winding 1 Current	The value currently being output to motor winding 1.
18	Winding 2 Current	The value currently being output to motor winding 2. Only valid for two or three phase motors.
19	Winding 3 Current	The value currently being output to motor winding 3. Only valid for three phase motors.
20	Analog input 0	The most recently read value from analog input 0.
21	Analog input 1	The most recently read value from analog input 1.
22	Analog input 2	The most recently read value from analog input 2.
23	Analog input 3	The most recently read value from analog input 3.
24	Analog input 4	The most recently read value from analog input 4.
25	Analog input 5	The most recently read value from analog input 5.
26	Analog input 6	The most recently read value from analog input 6.
27	Analog input 7	The most recently read value from analog input 7.

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.

For example; assume that the actual and command 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).

6.6.4 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.

6.6.5 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 <i>immediate</i> , 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 status register based trigger types, 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 source axis, bit number and bit state value are not used.
1	Update	The trace will start (stop) on the next update of the specified source axis. This trigger type does not use the bit number or 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.
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.
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.

6.6.6 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 effect 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).

6.6.7 Running Traces

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

1. 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. 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.
2. Set the trace period. The command **SetTracePeriod** allows an interval between trace samples to be specified.

3. Specify what data is to be stored. The command **SetTraceVariable** may be used to specify up to four variables to be stored each trace period. Make sure that variables 0 – (N-1) are used if N variables are to be stored, and that variable N is set to zero.
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.

6.7 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	Wrap-around	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	In positive limit switch	Positive over travel limit switch violation has occurred
6	In negative limit switch	Negative over-travel limit switch violation has occurred
7	Instruction error	Host instruction causes an error
11	Commutation error	<i>[MC2300 only]</i> 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 than one axis to be configured to generate interrupts at the same time, the chipset provides the command **GetInterruptingAxis**. 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.

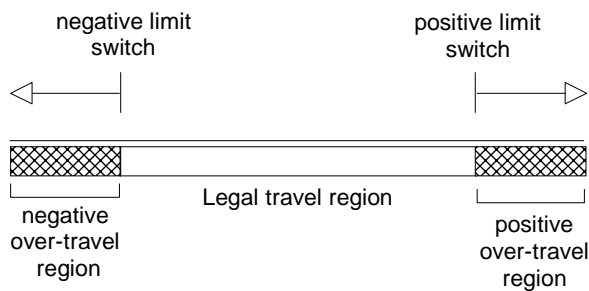
7 Hardware Signals

There are a number of signals which 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.

7.1 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.



The positive and negative switches are connected to CP chip inputs `PosLim1-4` and `NegLim1-4` respectively, to detect overtravel 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:

- 1) 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.
- 2) 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. 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 5.1.2 on instruction error 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 **SetLimitMode** 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.

7.2 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: <table border="0"> <tr> <td>ID</td> <td>register</td> </tr> <tr> <td>0</td> <td>None, the axis output pin will always be inactive</td> </tr> <tr> <td>1</td> <td>Event status register</td> </tr> <tr> <td>2</td> <td>Activity status register</td> </tr> <tr> <td>3</td> <td>Signal status register</td> </tr> <tr> <td>4-15</td> <td>Reserved, do not use</td> </tr> </table>	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
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.

7.3 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 Motor Interfacing

The Navigator chipsets support two types 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`.

8.1 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 8-3.

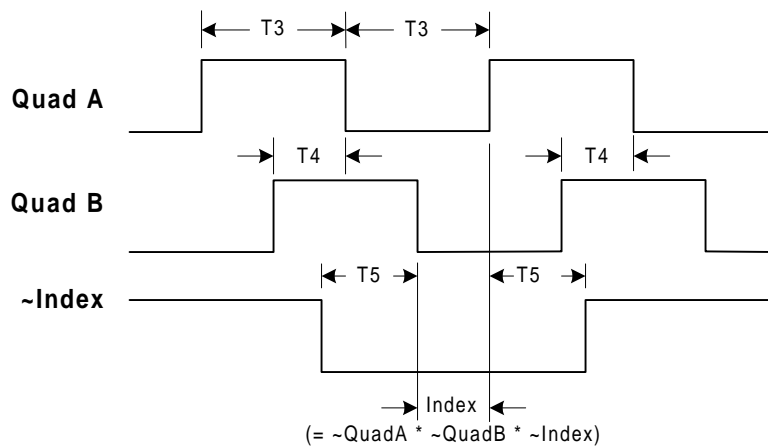


Figure 8-3. Quadrature Encoder Timing

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.

8.1.1 Actual Position Register

The chipset continually monitors the position feedback signals and accumulates a 32-bit position valued 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.

8.1.2 Digital Filtering

All encoder inputs as well 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.

8.1.3 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.

8.2 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 the following:

- Resolvers (after Resolver to Digital conversion)
- Absolute optical encoders
- Laser interferometers with parallel word read-out
- Incremental encoders with external quadrature decoder circuit
- 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.

8.2.1 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 $\frac{1}{2}$ 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 for the purpose of position capture the index and home signals, as well as the quadrature A and B signals are unused in this mode. Nevertheless for other non-position capture functions utilizing the signal status register these signals can still be used in the normal way. For example the se bits can be read (GetSignalStatus** command) or these bits can be used as breakpoint triggers, etc.**

8.2.2 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 according to the section of this manual entitled "Peripheral Device I/O" (section 8.3).

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. 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, and the values returned by the associated host peripheral circuitry will determine the actual position for axis 2 and 4.

8.3 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 Converters) signal output, parallel-word input, user-defined memory I/O, and serial port configuration input.

The bus consists of 5 control signals; \sim PeriphSlct, \sim Strobe, W/ \sim R, R/ \sim W, and \sim 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.

8.3.1 Peripheral Device Read

To perform a peripheral read the chipset will drive to a low level the \sim PeriphSlct signal, the \sim Strobe signal, and the W/ \sim 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.

8.3.2 Peripheral Device Write

To perform a peripheral write the chipset will drive to a low level the \sim PeriphSlct signal, the \sim Strobe signal, the R/ \sim W signal, and the \sim 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.

8.4 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:

Chipset family	Motors supported	# of phases per axis	available output formats
MC2100	Brushed Brushless permanent magnet (with external commutation)	1	sign magnitude PWM 50/50 PWM 16-bit DAC
MC2300	Brushless permanent magnet	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	standard speed pulse and direction high speed pulse and direction

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

4.1.1 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 8-2 shows some typical PWM output waveforms.

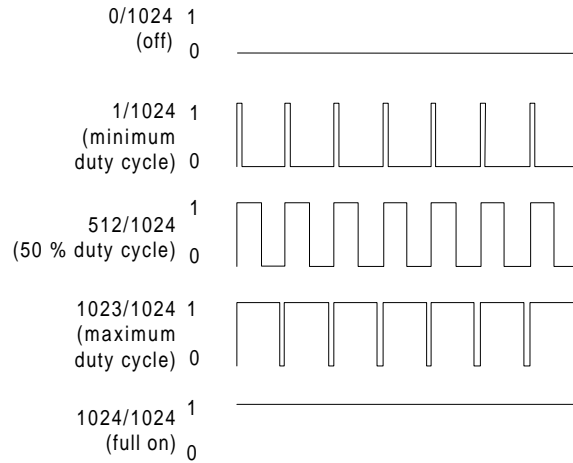


Figure 8-2. Typical PWM output signals

4.1.2 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.

4.1.3 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 μ sec, each axis will be written to once per 400 μ sec, with the writes separated by 100 μ sec

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 8.3.

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.

9 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 Figure 9-1:

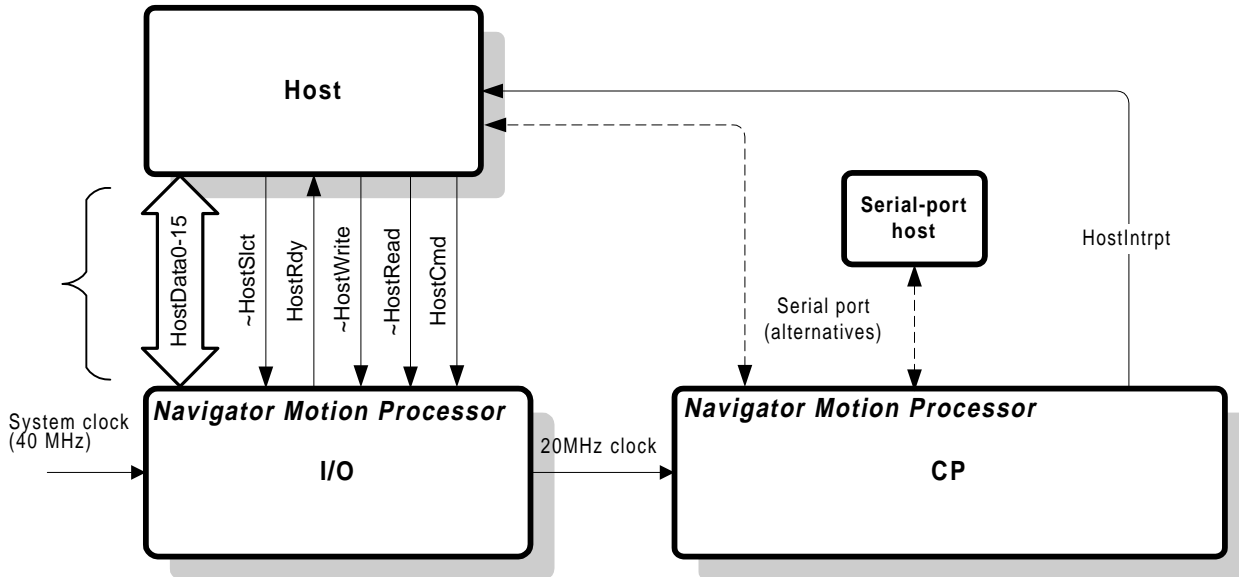


Figure 9-1. Host-chipset communications

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.

9.1 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.

9.1.1 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 effect 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, engineering 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 is executes a software program that can perform trace and display functions.

9.2 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 9.1).

9.2.1 Control signals

Five control signals synchronize communications through the parallel port¹: \sim HostSlct, HostRdy, \sim HostWrite, \sim HostRead, and HostCmd.

\simHostSlct	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.						
\simHostWrite	Set by host - when asserted low, allows a data transfer from the host to the chipset.						
\simHostRead	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 \sim HostRead and \sim HostWrite signals as follows: <table border="1"><tr><td><i>When</i></td><td><i>and HostCmd is</i></td></tr><tr><td>\simHostWrite is low</td><td>low: write data word to chipset high: write instruction word to chipset</td></tr><tr><td>\simHostRead is low</td><td>low: read data word from chipset high: read status byte from chipset (see Section9.2.3)</td></tr></table>	<i>When</i>	<i>and HostCmd is</i>	\sim HostWrite is low	low: write data word to chipset high: write instruction word to chipset	\sim HostRead is low	low: read data word from chipset high: read status byte from chipset (see Section9.2.3)
<i>When</i>	<i>and HostCmd is</i>						
\sim HostWrite is low	low: write data word to chipset high: write instruction word to chipset						
\sim HostRead is low	low: read data word from chipset high: read status byte from chipset (see Section9.2.3)						

9.2.2 Parallel port I/O operations

Using the five parallel port control signals of \sim HostSlct, HostRdy, \sim HostWrite, \sim 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 \sim HostSlct and \sim 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 \sim HostSlct, and \sim 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 \sim HostSlct, and \sim 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

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.

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.

At the beginning of each of these operations the \sim 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 \sim HostRdy does not need to be checked. For example after checking the \sim HostRdy for the high byte of any of these 2-byte transfers (*instruction word write, data word write, or data word read*) the \sim 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.

NOTE

Before any parallel host I/O operation is performed, the user must make sure that the \sim 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.

9.2.3 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 \sim HostRead low and HostCmd high, and reading the data bus. The resultant data word sent by the chipset contains the following information:

bit #	Description
0-12	unused, set to 0
13	Holds value of HostIOError signal (see section 9.3.1.2 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

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

9.3 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 read 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.

To illustrate command sequencing, a few command examples are listed below. For each of these commands 3 fundamental I/O operations are referenced; *the instruction word write*, the *data word write*, and the *data word read*. These same fundamental operations have already been defined in a previous section

ClearInterrupt // this command transfers no data to/from the chipset.

Operation #	operation	data value	Comments
1	instruction word write	00CAh	identifies this packet as a ClearInterrupt command
[2]*	[data word read]	[00CAh]	if a data word read is performed the returned value will be the checksum for the previous command, which in this case will be a value of 00CAh

*[brackets] indicate this operation is optional

SetVelocity Axis3, 12345678h // This command transfers two words of data to the chipset.

operation #	operation	data value	comments
1	instruction word write	0211h	identifies this packet as a SetVelocity command with an axis # of 3 (coded as a 2)
2	data word write	1234h	transfers a data word of 1234 (hex) to the chipset
3	data word write	5678h	transfers a data word of 5678 (hex) to the chipset
[4]	[data word read]	[6abd h]	if a data word read is performed the returned value will be the checksum for the previous command, which in this case will be a value of 6abd h

GetEventStatus Axis2 // This command reads one word of data from the chipse t.

operation #	operation	data value	Comments
1	instruction word write	0131h	identifies this packet as a GetEventStatus command with an axis # of 2 (coded as 1)
2	data word read	4307h	transfers a data word of 4307 (hex) to the host. The actual value of the event status register depends on the state of the axis, 4307 is used purely for example.
[3]	[data word read]	[4438]	if a data word read is performed the returned value will be the checksum for the previous command, which in this case will be a value of 68bdh.

GetBufferReadIndex **tracebuffer** // This command sends one word and reads two words of data.

operation #	operation	data value	Comments
1	instruction word write	00C7h	identifies this packet as a GetBufferReadIndex command
2	data word write	0h	transfers a data word of 0 to the chipset. This is the buffer Id. A 0 corresponds to the tracebuffer ID.
3	data word read	0001h	transfers a data word of 1 to the host. The actual value of the trace buffer index depends on the state of the chipset. 1 is used purely for example.
4	data word read	2345h	transfers a data word of 2345h to the host. The actual value of the trace buffer index depends on the state of the chipset. 2345h is used purely for example.
[5]	[data word read]	[240D]	if a data word read is performed the returned value will be the checksum for the previous command, which in this case will be a value of 240Dh.

3.1.1 Checksum

From the above sequences you will notice that 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.

3.1.2 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	<i>no error</i>	No error condition
1	<i>Navigator reset</i>	Default value of error code on reset or power-up
2	<i>invalid instruction</i>	Instruction is not valid in the current context, or an illegal instruction code has been detected.
3	<i>invalid axis</i>	The axis number contained in the upper bits of the instruction word is not supported by this chipset.
4	<i>invalid parameter</i>	The parameter value sent to the motion processor was out of its acceptable range.
5	<i>trace running</i>	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	<i>block bound exceeded</i>	<ol style="list-style-type: none"> 1. The value sent by SetBufferLength or SetBufferStart would create a memory block which extends beyond the allowed limits of 400h - 7FFFFFFh). 2. Either SetBufferReadIndex or SetBufferWriteIndex sent an index value greater than or equal to the block length.
8	<i>trace zero</i>	SetTraceStart Immediate was issued, but the length of the trace buffer is currently set to zero.
9	<i>bad checksum</i>	<i>(Serial port only)</i> The checksum compiled and returned by Navigator does not match that sent by the host.
Ah	<i>not primary port</i>	A prohibited instruction (one which can be executed only through the primary port) was issued through the diagnostic port.

9.4 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.

9.4.1 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 8.3 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 resistors (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	<i>Reserved</i>				
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.

9.4.2 Control signals

Three signals, *SrIXmt*, *SrIRcv*, and *SrIEnable* mediate serial transfers. *SrIXmt* encodes data transmitted from the CP to the host processor. *SrIRcv* receives data sent from the host to the CP. *SrIEnable* 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 *SrIEnable* 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)

9.4.3 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:

Address	One byte identifying which CP the command packet is being sent to. This field should always be zero in point-to-point mode.
Checksum	One byte value used to validate packet data. See description below.
Instruction code	Two byte instruction, sent upper byte (axis number) first. The command codes are the same as used in the parallel communication mode.
Data	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:

Status	Zero if the command was completed correctly, otherwise an error code specifying the nature of the error.
Checksum	A one-byte checksum value used to validate the packet's integrity. See details below.
Data	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.

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).

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.

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.

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 setup 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.

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.

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.

10 Using External Memory

10.1 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 can not 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 than 32K	An external page pointer must be used. The first 1K words of page zero are reserved for use by the CP.

10.1.1 Memory page pointer

If a memory address space of more than 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.

10.1.2 External memory signal decoding

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

The bus consists of 5 control signals; \sim RAMSlct, \sim Strobe, W/ \sim R, R/ \sim W, and \sim WriteEnbl. In addition there are 15 address lines (Addr0-Addr14) and 16 data lines (Data0-Data15).

10.1.3 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 \sim RAMSlct signal, the \sim Strobe signal, and the W/ \sim 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.

10.1.4 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 \sim RAMSlct signal, the \sim Strobe signal, the R/ \sim W signal, and the \sim 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.

10.1.5 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.

10.1.6 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.

NOTE: 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.

#

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