Danaher Motion

SERVO**STAR**® SC

User Guide

KOLLMORGEN

giving our customers freedom of design

Publication Number: KOL-1297
M-SS-006-0306
Firmware Version: 2.1.7

Record of Manual Revisions

ISSUE NO.	DATE	BRIEF DESCRIPTION OF REVISION
0	8/30/00	Preliminary issue for review
1	11/09/00	Initial release
2	01/19/01	Corrected text and updated format
3	06/01/01	Added commands for option card, added Appendix A and B and additional error codes
4	12/31/31	Added information for MODBUS TCP/IP and MODBUS RTU
5	04/22/02	Added information for DeviceNet
6	06/20/02	Corrected Regen information

Contents

OVERVIEW **1**

I/O 1
MOTION 2
Digital Position *2*
Digital Velocity *2*
Analog Velocity *2*
Units *3*
External Encoders *3*
Switching Modes *3*
Dynamic Changes 3
Static Changes 3
Stop Command 4
MOTIONSUITE AND BASIC MOVES 4
Communicating *4*
Ethernet Communications (Connector C6) 4
Serial Communications (Connector C7) 5
Program Debugging Tools *5*
API (Advanced Applications) *5*
MC-BASIC LANGUAGE COMPILER 5
Instructions *6*
Syntax *6*
Line Oriented *6*
Case Insensitive *6*
Program Declarations *6*
Subroutine Libraries (Advanced Applications) *7*
User-defined Functions (Advanced Applications) *8*
Recursive Function 8
Constants and Variables *9*
General Variables 9
Declaration 9
Variable Type 10
Scope 10
Data Types 10
Structures 11
NVRAM Variables 11
Expressions *11*
Algebraic Expressions 12
Automatic Conversion of Data Types 12
Logical Expressions 12
Operator Precedence 12
Operators *13*
Arithmetic Operators 13
Relational Operators 14
Logical Operators 14
Word-wise Operators 14
Bit-wise Operators 15
Math Functions *15*
String Functions *15*
System Commands *16*
Printing *17*
Synchronization 17
Hex and Decimal Mode 17
Flow Control *17*
Project *18*
Tasks 18
General Purpose (xxxx.Prg) 18

Configuration (Config.Prg) ... 20
AutoExec (Autoexec.Prg) ... 20
Create and Use Tasks ... 21
Loading the Program ... 21
Preemptive Multitasking and Priority Levels ... 21
Priority Level ... 21
Time Slice ... 22
Inter-Task Communications and Control Tasks ... 22
Monitoring Tasks from the Terminal ... 22
Relinquishing Resources ... 23
Multitasking (Advanced Applications) ... *23*
Event Handler ... *24*
Placing Event Handlers ... 24
OnEvent ... 24
Controlling OnEvent Occurrences ... 25
Events at Start-up ... 25
Program Flow and OnEvent ... 25
Error Handler ... *25*
Task Errors ... 26
OnError ... 27
Try/Finally ... 27
System Errors ... 28
OnSystemError ... 28
Error Print Level ... 28
Terminal Errors ... 29
Software WatchDog ... *29*
WatchDog Commands ... 29

SETTING UP AN AXIS ... 31

STARTING POSITION ... 31
Axis Setup Subroutine ... *32*
ACCELERATION AND DECELERATION ... 32
S-Curves ... *32*
SmoothFactor ... *32*
Acceleration Profile ... *32*
Position and Velocity ... *33*
ROTARY MODE ... 33
Rotary Mode Accuracy ... *33*
LIMITS ... 34
Generator Limits ... *34*
Realtime Limits ... *34*
Drive Limits ... *34*
POSITION LIMITS ... 34
PMin and PMax ... *34*
Position Error ... 35
PositionMinEnable/PositionMaxEnable ... *35*
Maximum Position Error ... 35
Position Error with Proportional Position Loop ... 36
Reducing Position Error ... 36
PositionErrorSettle ... 37
ACCELERATION FEED-FORWARD ... 37
TIMESETTLE ... 38
ISSETTLED ... 38
ISMOVING ... 38
AXIS VELOCITY LIMITS ... 39
VelocityOverspeed ... *39*
VelocityLim ... *39*
Axis Acceleration/Deceleration Limits ... *39*
AccelMax ... 39
DecelMax ... 39
CHECK YOUR SETTINGS ... 40
Motor Compensation ... *40*

SYS.CONMODE 40
ENABLING 40
MOTION FLAGS 41
LOOP TYPES 41
Position Loop *41*
Dual Loop *41*

SINGLE-AXIS MOTION 43

MOTION GENERATOR 43
CONDITIONS FOR MOTION 43
Attaching Motion to the Task *43*
Setting sys.conmode *44*
Hardware Enable *44*
Software Enable *44*
Motion Flag *44*
PROFILE 44
Update Rate *45*
PositionToGo *45*
MOTION BUFFERING 45
Override vs. Permanent *45*
Acceleration Profile *46*
JOGGING 46
Overriding Axis Properties In JOG *47*
STOP 47
Overriding Axis Properties In STOP *47*
PROCEED 47
MOVE 48
Point-to-Point (PFINAL) Moves *49*
Settling Time *49*
Starting Point-to-Point Moves *50*
Delay 50
StartType 50
Chaining Zero-Terminated Moves 51
Multi-Step Moves 51
Realtime Changes 53
Overriding Axis Properties in MOVE *53*
VELOCITY OVERRIDE 54
MOTION 54
Position Capture *54*
Homing *54*
Registration *55*
MASTER/SLAVE 55
Velocity Override *55*
Gearing *55*
Enabling and Disabling Gearing 56
GearRatio 56
Incremental Moves with Gearing 56
CAMming *58*
Key Features of CAMming 58
CAM and GearRatio 58
Incremental Moves with CAMming 58
CAM Tables 59
Interpolation 59
Incremental or Absolute 59
Net Motion in a CAM Table 60
CAM Table Length 60
Calculating Offsets in CAM Profiles 61
Rotary Master and/or Slave 61
Running CAM Cycles 61
Repeating the Table 61
CaSCycle 62

Linking Multiple CAM Tables ... 62
Linked CAMs and Cycle ... 62
Exhausting Cycles ... 62
Creating CAM Tables ... 62
Static CAM Tables ... 63
Building Static Tables in Windows ... 63
Create Point Pairs ... 63
Save to CSV ... 64
Building Static Tables in MC-BASIC ... 64
Globally Allocate CAM Storage ... 64
Create the CAM Data Table ... 64
Calculate the Points Pairs ... 65
Store the Table in Flash ... 65
Deleting CAMs (optional) ... 65
Review Entire Process ... 65
Dynamic CAM Tables ... 66
Operating CAMs ... 66
Allocate Space ... 66
Load CAM Data ... 66
Loading a Dynamic CAM ... 67
Initialize Elements of the CAM ... 67
Initialize the Axis ... 67
Enable the Axis ... 68
Read Dynamic Information ... 69
Axis Properties ... 69
CAM Properties ... 69

INPUT/OUTPUT AND PLS ... 71

GENERAL PURPOSE DIGITAL I/O ... 71
DEDICATED DIGITAL I/O ... 71
ANALOG I/O ... 71
PLS (PROGRAMMABLE LIMIT SWITCH) ... 72
Enable and Disable ... 72
Switch Positions ... 72
Repetition Interval ... 73
Polarity ... 74
Hysteresis ... 74
PLS Output State ... 74
PLS Set Up ... 74
Changing Polarity (Optional) ... 76
Disabling the PLS (Optional) ... 76
Deleting the PLS (Optional) ... 76

MODBUS PROTOCOLS ... 77

SLAVE ... 77
Connecting via TCP/IP to the Master ... 77
Connecting via RTU (Serial) to the Master ... 77
Setting the Communication Method ... 77
Changing Masters ... 78
Supported Functions ... 78
FC 0x11 Returned Data ... 78
Register Data Flow ... 78
Single Bit Data Flow ... 78
Data Reference ... 79
Inputs and Outputs ... 79
MODBUSSetBit ... 79
MODBUSSetRegister ... 79
MODBUSGetBit ... 79
MODBUSGetRegister ... 79
MODBUS TCP/IP Slave Example ... 80
MODBUS RTU Slave Example ... 80
MASTER ... 80

Memory Reference *81*
Register Data Flow *81*
Single Bit Data Flow *81*
MODBUS TCP/IP Master Example *82*
MODBUS RTU Master Example *82*

TROUBLESHOOTING **85**

CUSTOMER SUPPORT 85

APPENDIX A **A-1**

DIGITAL INCREMENTAL ENCODER TYPES A-1
Encoder Basics: A Review *A-1*
SERVOSTAR Encoder Types *A-4*
MENCTYPE 0 ..... A-4
MENCTYPE 1 ..... A-4
MENCTYPE 2 ..... A-4
MENCTYPE 3 ..... A-5
MENCTYPE 4 ..... A-5
MENCTYPE 5 ..... A-5
MENCTYPE 6 ..... A-5
Commutation Accuracy *A-7*
Physical Encoder Alignment *A-7*
MECNOFF *A-8*
System Phasing *A-8*
Troubleshooting *A-10*
Line Drivers, Receivers, and Terminations A-10
Design Considerations *A-10*
Reference Variables *A-11*

APPENDIX B **B-1**

RESISTIVE REGENERATION SIZING B-1
Energy Calculations *B-2*
Regeneration Calculations *B-2*
Determining Resistance Value B-2
Determining Dissipated Power B-3

APPENDIX C **C-1**

SUBROUTINE EXAMPLE C-1
CONTINUETASK EXAMPLE C-2
ONERROR EXAMPLE C-2
AXIS SETUP SUBROUTINE EXAMPLE C-3
ROTARY MODE EXAMPLE C-4
MOTION EXAMPLES C-5
Homing Example *C-5*
Registration Example *C-6*

APPENDIX D: DEVICE NET **D-1**

FUNCTIONALITY CHART D-1
OBJECT MODEL D-2
POSITION CONTROLLER DATA TYPES D-4
Supported Services *D-4*
Data Types *D-4*
POSITION CONTROLLER SUPERVISOR OBJECT CLASS (ID=36) D-5
Error Codes *D-5*
SUPERVISOR ATTRIBUTES D-6
POSITION CONTROLLER OBJECT CLASS (ID=37) D-13
Error Codes *D-13*
BLOCK SEQUENCER OBJECT CLASS (ID=38) D-25
COMMAND BLOCK OBJECT CLASS (ID=39) D-27
Command 01 – Modify Attribute *D-28*

Command 02– Wait Equals Command *D-30*
Command 03– Conditional Link Greater Than Command *D-32*
Command 04– Conditional Link Less Than Command *D-34*
Command 06– Delay Command *D-36*
Command 07– Trajectory Command *D-37*
Command 08– Trajectory Command and Wait *D-39*
Command 09– Velocity Change Command *D-41*
Command 10– Go To Home Command *D-42*
Command 12– Go To Registration Command *D-43*
POLLED I/O COMMAND ASSEMBLIES D-44
Command Assembly 1: Target Position *D-45*
Command Assembly 2 – Target Velocity *D-47*
Command Assembly 3 – Acceleration *D-48*
Command Assembly 4 – Deceleration *D-49*
Command Assembly 26 – Position Controller Supervisor Attribute *D-50*
Command Assembly 27 – Position Controller Attribute *D-51*
Command Assembly 28 – Block Sequencer Attribute *D-52*
Command Assembly 30 – Command Block Attribute *D-53*
POLLED I/O RESPONSE ASSEMBLIES D-54
Response Assembly 1 – Actual Position *D-54*
Response Assembly 2 – Commanded Position *D-56*
Response Assembly 3 – Actual Velocity *D-57*
Response Assembly 4 – Commanded Velocity *D-58*
Response Assembly 6 – Home Position *D-59*
Response Assembly 8 – Registration Position *D-60*
Response Assembly 20 – Command/Response Error *D-61*
Response Assembly 26 – Position Controller Supervisor Attribute *D-63*
Response Assembly 27 – Position Controller Attribute *D-64*
Response Assembly 28 – Block Sequencer Attribute *D-65*
Response Assembly 30 – Command Block Attribute *D-66*
IDENTITY OBJECT CLASS (0X01) D-67
MESSAGE ROUTER CLASS (0X02) D-68
DEVICENET OBJECT (0X03) D-69
ASSEMBLY OBJECT (0X04) D-70
CONNECTION OBJECT (0X05) D-71
ACK HANDLER OBJECT (0X2B) D-72

OVERVIEW

The SERVO**STAR**® SC is shipped with a *SERVO**STAR**® SC Setup Guide*. The Setup Guide contains all the information necessary to get your system up and running. For detailed installation instructions (including screen shots), refer to the *SERVO**STAR**® SC Installation Manual.* This User's manual describes how to use the SERVO**STAR** SC product and MC-BASIC commands, troubleshooting tips, and error codes The*SERVO**STAR**® SC Reference Manual* contains the entire list of variables and commands. In order to execute the examples described in this manual, you must at least have a SERVO**STAR** SC and BASIC Moves Development Studio or MotionSuite installed on your host computer. When you complete this manual, you should feel comfortable using all the features available in the SERVO**STAR** SC.

The SERVO**STAR** SC is Kollmorgen's single-axis controller. The SERVO**STAR** SC provides fully digital current, velocity, and position loops. It allows automatic control loop tuning and advanced velocity control alogrithms. Its advanced patented sinewave communication technology provides smooth, precise low-speed control and high-speed performance. Accurate torque control is achieved from precision balanced current loops with closed-loop sensors. The patented torque angle control enhances motor performance. The SERVO**STAR** SC provides velocity loop bandwidths up to 400 Hz. The feedback can be resolver, encoder, or sine encoder. A secondary encoder feedback can be used to close a dual loop around the load or as an input for master/slave or pulse-and-direction operation.

The SERVO**STAR** SC has large on-board memory, with ample capacity for expansion and enough space for your program. There is also an on-board Flash disk for storage of your programs.

The SERVO**STAR** SC has the features and performance required of today's single-axis controllers and is designed for easy integration into motion control systems and for simplicity of use. The key benefits of the product are:

- **Motion Components Guaranteed to Work Together**
  Kollmorgen provides a complete motion control system, including controller, drives, and motors. All the components supplied by Kollmorgen are tested and guaranteed to work together.
- **Complete Digital System at an Affordable Price**
  The SERVO**STAR** SC is fully digital, including communication with the drives. The analog interface is eliminated, but the price is competitive with analog systems.
- **Windows-Based Software Supports Microsoft Visual Basic™, Visual C++™, and other Popular Languages**
  The SERVO**STAR** SC product family includes the Kollmorgen API, which allows Windows NT® application integration with the SERVO**STAR** SC controller. Communication uses dual-port RAM, so the process is fast and reliable.

I/O

The SERVO**STAR** SC provides three "fast" dedicated inputs (HOME, LIMIT, CAPTURE), a dedicated remote input (enable/disable), a dedicated motion input (allow/prohibit), and sixteen fully-isolated inputs. It also has a dedicated status output (ready/active) and eight fully-isolated outputs as standard features. In addition, it has two 14-bit resolution analog inputs and two 12-bit resolution analog outputs.

Motion

The SERVO**STAR** SC profile provides sine acceleration plus automatic or manual jerk control and allows trapezoidal velocity for rapid motions. The SERVO**STAR** SC supports jogging, incremental or absolute moves, non-zero final velocity, and advanced stop and proceed commands. The SERVO**STAR** SC allows on-the-fly changes between operational modes. The SERVO**STAR** SC operation modes are:

1. Digital position
2. Digital velocity
3. Analog velocity

Digital Position

In position mode, the control loop is closed using the position feedback of the motor. The SERVO**STAR** SC control always tries to minimize the difference between the feedback and the command position. Initially, after start-up, the system is in digital position mode. In digital position mode, the axis can be moved only in following states. The following states are:

Move-command execution
Geared
CAMmed

If you enter:

```
Move 10000
```

The axis moves to position 1000 and switches to digital position mode. During command execution, all the usual checking is performed regularly. This includes:

Following error
Velocity over-speed
Position limits
Current limit

Digital Velocity

In velocity mode, the control loop is closed using the velocity feedback and command of the motor. The position command and feedback are calculated but not used in the control loop. This mode is used in applications where continuous movements without respect to position are required. In digital velocity mode, the axis is moved only with the jog command execution and in static (no motion) states. During command execution, only the following checking is performed regularly:

Velocity over-speed
Position limits
Current limit

Analog Velocity

Analog velocity is similar to digital velocity mode except that the velocity command is generated from an external source (i. e., the voltage source connected to ANIN1). There is also the AnalogVelocityScale parameter, which translates the analog input (milli-Volts) into the RPM of the command velocity. The position command is not generated.

Units

Units are fixed at:

Position	Count
Velocity	rpm
Acceleration	rpm/sec
Time	ms

External Encoders

The SERVO**STAR** SC supports external encoders (encoders not connected to the axis controlled by the SERVO**STAR** SC). These encoders are connected to the system using the external encoder input of any SERVO**STAR** SC in the system.

Switching Modes

Switching modes on an axis is performed with dynamic changes, static changes, or using the STOP command.

DYNAMIC CHANGES

If the axis is moving, operational mode can only be changed using immediate commands. This allows for on-the-fly changing. You can issue a command to make the change immediately. This change occurs after 5 ms, or 5 sample times. For example:

```
Move 1000000              'Move to a distance point
Sleep 3000                'Wait 3 sec after the motion started

Jog 100 StartType = 1     'Change the operational mode immediately
```

You can also issue a STARTTYPE super immediate command. If the STARTTYPE super immediate command is used, the change occurs within the sample time executing. If you wish to use STARTTYPE super immediate, you enter:

```
Jog 100 StartType = 5   'Change the operational mode super immediately
```

Another example:

```
AnalgoVelocityScale = 10   'Output 10 RPM for each mV on the analog input
Opmode 1                   'Start the analog velocity mode
Sleep 5000                 'Wait 5 sec
Move 10000 StartType = 1   'Change operational mode to digital position
```

In this case, STARTTYPE immediate must be used.

STATIC CHANGES

When the motor is at rest, the operational mode can be changed implicitly or explicitly. Implicit operation mode change allows you to issue one of two motion commands:

MOVE – enters the system implicitly to digital position mode
JOG – enters the system implicitly to digital velocity mode

Explicit operation mode change is where the operational mode is changed by issuing the OPMODE command. If the OPMODE command is issued during a motion, the mode change occurs at the end of the current motion. Use caution with this command as this change may never occur if the motion is endless (JOG). If the OPMODE is entered when no motion is executing, the operation mode is changed immediately.

The only exception is the analog mode. If an OPMODE command is entered while the system is in analog mode, the motor is stopped immediately and the system is changed to digital mode (position or velocity). For example:

```
Move 10000        'digital position mode
Opmode dvmode     'change to digital velocity mode
```

You could also issue this same command using:

```
Move 10000          'digital position mode
Opmode0             'change to digital velocity mode
```

Since this command was entered during the motion, it is executed at the end of the current motion. Refer to the OPMODE command for additional information.

STOP COMMAND

In any of the digital modes, the STOP command reduces the velocity to zero (DECSTOP) and the system stays in its original operational mode. This means, if there was a JOG command executing (means digital velocity mode) and the STOP command is issued, the system stops the motion but the operational mode remains in digital velocity mode.

In analog velocity mode, the STOP command immediately changes the operational mode to digital velocity mode and reduces the current velocity to zero.

MotionSuite and BASIC Moves

MotionSuite's Program Development and BASIC Moves Development Studio (BMDS) provide Windows-based project control for each application and all the tools you need for developing and debugging your application. Both applications support development for multitasking as well as numerous tools and wizards to simplify programming the SERVO**STAR** SC.

These applications provide the user interface to the SERVO**STAR** SC motion control system as well as modern debugging features (such as task control by visually setting breakpoints, watch variables, and single stepping). They automate the display of data recorded on the SERVO**STAR** SC during operation.

When you start either of these applications, your first action is to select a method for communicating with your SERVO**STAR** SC. Select either Serial Port or Ethernet (the method configured for your system). For more information concerning communications with the SERVO**STAR** SC, refer to the *SERVO**STAR®** SC Installation Manual*.

The online Installation Help is especially convenient for viewing this information.

Communicating

Communicating with a SERVO**STAR** SC is not automatic. Assuming you have properly configured the communication method during installation of either MotionSuite or BASIC Moves on your host computer, there are still some operating procedures you may need to execute. Refer to the *SERVO**STAR®** SC Installation Manual* for additional information.

ETHERNET COMMUNICATIONS (CONNECTOR C6)

If you configured Ethernet communications, subsequent communication with the SERVO**STAR** SC is automatically enabled and no further intervention is needed unless you change the Ethernet network environment. If your network environment changes, you may need to edit the IP address file.

Be sure to contact your system administrator before making any changes to your IP address file. Refer to the *SERVO**STAR®** SC Installation Manual* ***for additional information.***

SERIAL COMMUNICATIONS (CONNECTOR C7)

If you configured serial communications, you must explicitly enable serial communications using Dial Up Networking each time you start MotionSuite or BASIC Moves and after cycling power off or on either the SERVO**STAR** SC or the host computer. If you leave the serial connection up, you can repeatedly enter and leave these applications without disturbing the connection. Refer to the *SERVO**STAR®** SC Installation Manual* for additional information.

Program Debugging Tools

MotionSuite's Program Development and BASIC Moves Development Studio has several tools to help debug a program:

Message Log Window– provides information on errors that occur when loading a program to RAM (getting ready to run).

ErrorHistory Property – Use this command in the terminal window to view current and previous errors.

ServoError Property – Use this command in the Terminal Window to view any current servo errors that have not been cleared.

Single Step – Use the buttons on the toolbar to single step through the program or task.

Insert Stops – Use the buttons on the toolbar to insert points in a task at which program execution stops.

Watch Window – Allows monitoring of many SERVO**STAR** SC properties in realtime.

Try Property – Allows code inserted in the task(s) to take specific action based on the errors that occur during program execution.

API (Advanced Applications)

The Kollmorgen API software package allows you to communicate with the SERVO**STAR** SC with such popular programming languages as Visual Basic. The API provides complete access to all the elements of your system across dual-port RAM. See the *API Reference Manual* for more information about Kollmorgen's API.

MC-BASIC Language Compiler

The SERVO**STAR** SC is fast (executing many command lines in under 100 microseconds). One reason is because the language is semi-compiled.

The SERVO**STAR** SC is programmed in MC-BASIC, a version of the BASIC programming language enhanced for multitasking motion control. MC-BASIC is true BASIC. If you are familiar with BASIC, you already know much of MC-BASIC. It has familiar commands such as "FOR/NEXT", "IF/THEN", "PRINTUSING", and common string functions such as "CHR$", "INSTR", "MID$", and "STRING$". It has arrays (up to 10 dimensions) and full support for double-precision floating-point math. In addition, MC-BASIC is extended to provide support for functions that motion systems require: point to point moves, circular and linear interpolation, CAMming, and gearing, multitasking and event-driven programs. Refer to the *SERVO**STAR®** SC Reference Manual* for a full listing with examples of the commands and functions.

Gearing and CAMming have the flexibility required for modern motion control. This can only be accomplished through encoder feedback.

CAMming links the master and slave axis together by a CAM table. CAMming has all the features of gearing, and the table can have any number of points. The CAM points are x-y format so spacing is provided independently. Multiple CAM tables are linked together, allowing you to build complex CAM profiles from simpler tables. There is a cycle counter that lets you specify how many cycles to run a CAM before ending.

The SERVO**STAR** SC supports up to 256 tasks running at up to 16 different priority levels for multitasking. Each task can generate "OnEvent(s)," for code that you want executed when events occur, such as switches tripping, a motor crossing a position, or any combination of factors.

Instructions

Instructions are the building blocks of BASIC. (In this manual terms "instruction" and "command" are used interchangeably.) Instructions are used to set variables, call functions, control program flow, and start processes such as events and motion. Instructions can be comments, assignments, memory allocation, flow control, task control, and motion.

Comments	Comments allow you to document your program.
Memory Allocation	Memory allocation instructs the controller to set aside space for variables.
Assignment	Assignment instructions assign a new value to a variable.
Print	To query a variable or expression from the terminal window, use the “PRINT” or “?” command. MC-BASIC also provides the standard "PRINTUSING" ("PRINTU") command for formatted printing.
Flow Control	Commands for flow control change the way your program is executed. Without flow control, program execution would be limited to processing the line immediately following the current command.
Inter-Task Control	MC-BASIC is a multitasking language in which many tasks can run concurrently.
Persistence	Most commands are started and finished immediately.

Syntax

Syntax is the set of rules that must be observed to construct a legal command (command(s) the SERVO**STAR** SC recognizes). The following notation is used in the syntax:

[ ] indicates the contents are required
{ } indicates the contents are optional for the command

Example lines of text are shown in Courier type font and with a border:

```
X = 1
```

Line Oriented

MC-BASIC is line-oriented. The end of the line indicates the end of the instruction. White space (spaces and tabs within the statement line and blank lines), is ignored by the BASIC interpreter. You can freely use indentation to delineate block structures in your program code for easier readability. The maximum allowed line length is 80 characters.

Case Insensitive

MC-BASIC is case insensitive. Commands, variable names, filenames, and task names may be written using either upper case or lower case letters. Strings are the only exception. They are case-sensitive. For example:

```
? "hello" = "HELLO"
```

Returns False or 0 while:

```
? "HELLO" = "HELLO"
```

Returns True or 1.

Program Declarations

You must declare the start of programs and subroutines. For programs, use the Program command. Use Sub or Subroutine for subroutines. (For an example, refer to Appendix C).

Program	The Program command marks the boundary between the variable declaration section and the main program. Each task has only one Program command.

Subroutines You can pass parameters (either scalar or array) to the subroutine, which can then be used in the code of the subroutine. Subroutines may be recursive (a subroutine that calls itself).

You can define as many Subroutines as you need.

Subroutine Libraries (Advanced Applications)

As you develop programs for a variety of applications, you will find that there are some subroutines that you use repeatedly in the programs you write. You can collect such subroutines (and user-defined functions) into a library file, then when you program a task, import the .LIB file at the beginning of the program and you can then call any of its defined subroutines (functions). An MC-BASIC library is an ASCII file containing only the sub-program's code; the file does not have a main program part. The names of the library files must have the extension ".LIB". The format of a library file is:

```
Declaration of static variables
...Etc.
{PUBLIC}SUB <sub_name_1> etc…
       {Declaration of variables local to the sub-program}
       {sub-program code}
END SUB

…etc …

{PUBLIC} SUB <sub_name_n> etc…
       {Declaration of variables local to the sub-program}
       {sub-program code}
END SUB
```

The {PUBLIC} keyword allows a subroutine to be called from within any task. These subroutines are visible from outside the scope of the library. Sub-routines declared without the {PUBLIC} keyword can only be used inside the scope of the library. They are, in fact, PRIVATE library functions. For example:

```
PUBLIC  SUB public_sub(…etc…)
…etc…
END SUB


SUB private_sub(…etc…)
…etc…
END SUB
```

The subroutine "private_sub" can be used only inside the library. For example, it can be called from another subroutine of that library. The "public_sub" can be used in any program file by importing the library into it. The scope of the library is the task that imports it. However, you can import the library into each of multiple tasks.

A library file can be used like any other program file. You can send a library file to the Flash disk and you can retrieve it or delete it. In order to use a library file in a program, the library must first be loaded into memory; the syntax for loading is as follows:

```
LOAD MyLibrary.lib
```

The syntax for unloading the library is:

```
UNLOAD MyLibrary.lib
```

If a program task or a library task references a subroutine in a library, then the program task or the library task must be unloaded from memory before the library can be unloaded.

After a library is loaded in memory, it must then be imported into a program. The IMPORT command must be placed at the beginning of the program code before any other command or declaration. The syntax for the IMPORT command is:

```
IMPORT MyLibrary.lib
```

The syntax for calling a library subroutine or function from a task is the same as the syntax for calling a local subroutine from the task. You do not need to specify which library file contains the subroutine you want to call. You can not import two library files containing the same library function name.

User-defined Functions (Advanced Applications)

MC-BASIC allows you to define functions that can be used in programs in the same manner as using BASIC's pre-defined functions. User-defined functions can be composed with multiple lines and can be recursive (calls itself). Unlike BASIC's system functions, the scope of user-defined functions is limited to the task in which it is defined.

Functions are different from subroutines in one respect: functions always return a value to the task that called the function. Otherwise, functions and subroutines use the same syntax and follow the same rules of application and behavior. Because functions return a value, you can use functions in an expression:

```
tangent = sin(x) / cos(x)
```

Or

```
If  (Sgn(x) Or Sgn(y)) Then…
```

The syntax for defining a function is:

```
Function <name> ({{ByVal}<p_1>as <type_1>}…{,{ByVal}<p_n> as type_n>}) As<function type>
       { local variable declaration }
       { function code }
END function
```

You can pass parameters (either scalar or array) to the function, which are then used in the code of the function. In the declaration line for the function (FUNCTION <*name*>…), you declare the variable names and types of the parameters that you want to pass.. Parameters can be passed either by reference or by value (*ByVal*); the default method is to pass parameters by reference. When you pass a variable (whether the variable is local to the task or is global) by reference, you actually pass the address of the variable to the function, which can then change the value of the variable (if the code of the function is written to do this). When you pass a variable by value (*ByVal*) a copy of the value of the local variable is passed to the function, consequently, the function can not change the value of the local variable. Arrays can be passed only by reference.

To set up the return value, somewhere in the code of the function, assign the value that you want to return to a virtual variable that has the same name as the function. You don't declare the variable, but this is the mechanism that the function uses to obtain the return value. If the function's code includes conditional statements, assignments to the function variable can be made in multiple locations in the function code.

There is no explicit limit on the number of functions allowed in a task. All functions must be located following the main program, and must be contained wholly outside the main program.

RECURSIVE FUNCTION

As noted above, a function can be defined to be recursive (calls itself). For example, the mathematical function, N!, can be evaluated in a recursive function. Consider:

```
N! = N * (N-1)!
```

The following example defines a recursive function to calculate the value of N!:

```
FUNCTION Factorial(By Val N as Long) as Double
'By declaring parameter N to be type long, this truncates floating-point numbers to integers.
The function returns a Double value.

      If N < 3  then       'This statement stops the recursion
            Factorial = N    '0! = 0, 1! = 1, 2! = 2
      Else
            Factorial = N * Factorial(N-1)   'Recursive statement
      End if

END FUNCTION
```

When writing recursive functions, you must have an If statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function never returns once you call it.

Constants and Variables

Constants and variables are used throughout MC-BASIC. There are two types of variables used for the SERVO**STAR** SC. One type (general variables) are loaded in RAM. The second type (NVRAM) is stored in NVRAM. General variables are discussed first.

Names All names in the SERVO**STAR** SC must start with an alphabetical character (a-z, A-Z) and may be followed with up to 31 alphabetical characters, numbers (0-9) and underscores ("_"). Keywords may not be used as names.

Constants Constants are numbers written as ordinary text characters. Constants can be written in decimal or in hexadecimal (HEX).

Literal Constants The SERVO**STAR** SC provides numerous "literal constants" or reserved words that have a fixed value. You can use these constants anywhere in your program to make your code more intuitive and easier to read. Some of the constants are listed in the table below.

CAM = 2	Used for the SLAVE property.
CLEARMOTION or CMOT = 3	Used in the PROCEEDTYPE property.
CONTINUE or CONT = 1	Used in the PROCEEDTYPE property.
ENDMOTION = 3	Used in the STOPTYPE property.
FALSE = 0	
GEAR = 1	Used in the SLAVE property.
GENERATORCOMPLETE or GCOM = 3	Used for the STARTTYPE property.
IMMEDIATE or IMMED = 1	Normally used in the STARTTYPE and STOPTYPE properties.
INPOSITION or INPOS= 2	Normally used in the STARTTYPE property.
LINEAR = 0	Used in setting motion type.
NEXTMOTION or NMOT = 2	Used in the PROCEEDTYPE property.
OFF = 0	
ON = 1	
ONPATH = 2	Used in the STOPTYPE property.
PI = 3.141592653590001	
ROTARY = 1	Used in setting motion type.
SUPERIMMEDIATE or SIMM = 5	Normally used in the STARTTYPE and STOPTYPE properties.
SYNC = 4	Used for the STARTTYPE property
TRUE = 1	

GENERAL VARIABLES

General variables are stored in RAM. Before you can use these variables, you must declare them.

DECLARATION

In the declaration, you define the scope and variable type.

VARIABLE TYPE

MC-BASIC supports Long for integer values, Double for floating-point values, and String for ASCII character strings. Besides these BASIC types, MC-BASIC also supports Structure type variables such as CAM, Group, and PLS.

SCOPE

Scope defines how widely a variable can be accessed. The broadest scope is "global." A variable with global scope can be read from any part of the system software. Other scopes are more restrictive, limiting access of variables to certain sections of code.

MC-BASIC supports three scopes: global, task, and local scope. Task scope means it can be read from or written to anywhere within the task where it is defined, but not from outside that task. Local can only be used within a program or subprogram, each of which is a section of tasks. The scope of a variable is implicitly defined when a variable is declared using the keywords Common, Shared, and Dim.

Global Scope Variables with global scope must be defined within the system configuration task, Config.Prg or from the terminal window of BMDS. To declare a variable of global scope use the Common Shared instruction.

Task Scope Variables with task scope are defined within a task. To declare a variable of task scope use the Dim Shared instruction.

Local Scope Variables with local scope are defined and used within a program, a subroutine, or a function. To declare a variable of local scope use the Dim instruction.

DATA TYPES

Numeric Data Types MC-BASIC has two data types you can declare when you define a variable: Long and Double. Long is the only integer form supported by MC-BASIC. Long and Double are MC-BASIC's "primitive data types".

Type	Description	Range
Long	32 bit signed integer	-2,147,483,648 (MinInteger) to 2,147,483,647 (MaxInteger)
Double	Double precision floating point (about 16 places of accuracy)	±1.79769313486223157 E+308

String Data Types MC-BASIC provides the string data type consisting of a "string" of ASCII-coded characters. A full complement of string functions is provided to allow creation and modification of strings.

Type	Description	Range
String	ASCII character string (no practical limit to length of string)	0 to 128 (ASCII code)

Logical Data Types Logical (Boolean) variables have one of two values: true or false. MC-BASIC uses type Long to support logical expressions (0 = False). Usually, 1 = True, although any non-zero value is taken to be true. Boolean expressions are used in conditional statements such as IF…THEN, and with Boolean operators like AND, OR, and NOT.

MC-BASIC uses integers to support Boolean operations.

Arrays You can create arrays with any data type. Arrays may have from 1 to 10 dimensions; the maximum number of elements in any dimension is 32 767. Array dimensions always start at 1.

Arrays can be of any data type.

STRUCTURES

Many MC-BASIC variables are grouped or "encapsulated" in structures. A structure is a group of data elements contained under one name. Structures simplify programming by grouping data together. For example, an axis is a data structure. An axis has many data elements including position command, position feedback, velocity command, and so on. An axis would have data elements such as:

PositionFeedback
VelocityCommand
VelocityFeedback
PositionFinal

A data element of a structure can be used like any other data element. You can combine them in expressions, use them in function calls, and so on. For example, the following statements are all legal:

```
x = PositionFeedback                          'set x to the position
y = PositionCommand + PositionCommand         'combine two commands
z = Log(Abs(VelocityFeedback))                'this isn't done often.
```

Many data elements are organized in structures such as those related to an axis or to a group of axes. These structures are predefined by MC-BASIC. You cannot define new structure types in your program.

NVRAM VARIABLES

These variables are stored in NVRAM. This means NVRAM variables cannot be declared or given a name or dimension. The names of these variables are fixed and can be either Long or Double type. You can specify the number of elements required for the Long or Double NVRAM vector and change it dynamically. This dynamic change reformats the NVRAM memory so that all the data previously stored, is cleared from memory.

The Long and Double NVRAM variable names are: LNVRAM and DNVRAM. These are reserved keywords that cannot be used for system variable names.

Refer to the specific NVRAM variables (DNVRAM, DNVRAMSize, NVRAMFormat, LNVRAM, and LNVRAMSize) for additional information.

Expressions

Expressions are combinations of variables, constants, functions, and operators that calculate to a value. Operators specify how to combine the variables and constants. Expressions are used in almost all commands. There are two types of expressions: algebraic and logical (or "Boolean").

An expression has one of the two forms:

[operand] [binary operator] [operand]
[unary operator] [operand]

An operand can be a constant (123.4) or a variable name (X), or it can be another expression. Most operators are binary—they take two operands—but some are unary, taking only one operand.

ALGEBRAIC EXPRESSIONS

Algebraic expressions are combinations of data and algebraic operators. The results of all algebraic operations are converted to double-precision floating point after executing the expression. For example, open the terminal window and type in the following line:

```
Common Shared B1 as Long
```

While "B1" might be an integer, "B1+1" is a double precision number. So:

```
? B1
```

Prints B1 as an integer, but:

```
? B1 + 1
```

Prints as a double precision number because all algebraic expressions are converted to double.

AUTOMATIC CONVERSION OF DATA TYPES

If the assignment of an expression is an integer, the expression is still evaluated in double precision. For example:

```
Common Shared B1 as Long
B1 = 6.33 * 2.79
```

The value of the expression (6.33 * 2.79) is converted from double to long when the value (17) is copied into I. In the conversion to long, the number is truncated to the integer portion of the floating-point value.

The SERVO**STAR** SC warns you if an integer has overflowed. For example:

```
B1 = 2 ^31
```

Generates an error.

LOGICAL EXPRESSIONS

Logical expressions are combinations of data and logical operators. For example, "X1 And X2" is a logical expression. The data and results of logical elements are type Long. Double variables are not allowed in logical expressions. The terms "logical" and "Boolean" may be considered interchangeable.

Logical expressions are evaluated from left to right. They are evaluated only far enough to determine the result. In:

```
A And B And C
```

If A is 0 (false), the result is seen to be false without evaluating B or C.

OPERATOR PRECEDENCE

Precedence dictates which operator is executed first. If two operators appear in an expression, the operator with higher precedence is executed first. For example, multiplication has a higher precedence than addition. The expression 1+2*3 is evaluated as 7 (2 * 3 first), not 9. The minus sign in the math-operators table (below) is sometimes confusing since it shows up twice in the table: once with a high precedence (unary negation) and once with low precedence subtraction (binary minus). This difference in precedence is intuitive: 5 * - 6 only makes sense if the negation is executed first. Finally, any time you need to change the standard precedence, you can use parentheses. Parentheses have the highest precedence of all.

Operators

The tables shown below list the different types of operators, and their relative precedence. Arithmetic operators have the highest precedence, followed by Relational, Logical and Bit-wise operators. The relative precedence of the operators within each group is noted in the tables. Parentheses may be used to force a different order of expression processing because parentheses have the highest precedence. The "+" operator can be used to concatenate strings, and the "=" operator can be used to copy strings. The other operators have no applicability with strings.

ARITHMETIC OPERATORS

Operation	Symbol	Relative Precedence	Unary/Binary	Operand Type
Parentheses	()	1	Not applicable	Double, long, string
Exponentiation	^	1	Binary	Double, long
Negation/N/A minus	-	2	Unary	Double, long
Multiplication	*	3	Binary	Double, long
Division	/	3	Binary	Double, long
Modulus	MOD	4	Binary	Double
Addition	+	5	Binary	Double, long, string
Subtraction	-	5	Binary	Double, long

Negation is simply placing a minus sign in front of a constant or variable to invert its arithmetic sign. Modulo division produces the "remainder" of an addition. For example, 82 Mod 5 is equal to 2—the remainder of 82/5. If the first operand is negative then the resultant value is negative, -2. The sign of the first operand determines the sign of the result and the sign of the second operand has no effect.

The relative precedence of exponentiation and negation may seem somewhat counter-intuitive. The following examples illustrate the actual behavior.

```
Common shared D as double
D = -1
? D^2
1
? -1^2
-1
```

In the second example, exponentiation is performed first then negation is performed.

In this example, the "=" sign is taken as a relational operator (i.e., "Is D = -1?"). The relation is True, or 1.

```
Common shared D as double
D = -1
? D = -1
1
```

In this example also, the "=" sign is taken as a relational operator (i.e., "Is D^2 = -1^2?"). Again, the exponentiation is performed first, and then the relation is evaluated as False, or 0. The "+" operator, when used with string variables, performs concatenation of strings.

```
? D^2 = -1^2
0
```

RELATIONAL OPERATORS

Relational operators provide a method whereby two values may be compared. The result of the comparison is either TRUE (1) or FALSE (0). The operators are listed below; all Relational Operators have the same level of precedence, and relational operators have lower precedence than arithmetic operators. Relational operators may be used with any type of variable or value, including strings. String comparisons are made character-by-character, starting with the first character in each string, thus the string “de” is greater than the string “abc”. Character comparisons are made according to the value of the ASCII character codes for the respective characters.

Operation	Symbol	Relative Precedence	Binary/Unary	Operand Type
Parentheses	( )	1	Not applicable	Double, long, string
Equality	=	1	Binary	Double, long, string
Inequality	<>	1	Binary	Double, long, string
Less than	<	1	Binary	Double, long, string
Greater than	>	1	Binary	Double, long, string
Less than or equal to	<=	1	Binary	Double, long, string
Greater than or equal to	>=	1	Binary	Double, long, string

LOGICAL OPERATORS

Logical operators use logical (boolean) math: "And-ing" and "Or-ing" to form a result. There are two types of logical operators: Word-wise and Bit-wise.

WORD-WISE OPERATORS

Word-wise operators produce a single value of either "true" or "false". Logical operators have lower precedence than Relational operators. They may only be used on integer operands (including the results of relational expressions). In this manual, unless otherwise stated, "logical" can be assumed to be word-wise.

Word-wise Logical Operator	Symbol	Relative Precedence	Unary/Binary	Operand Type
Not	Not	1	Unary	long
And	And	2	Binary	long
Or	Or	3	Binary	long
Exclusive Or	Xor	4	Binary	long

In Logical operations, every value that is not 0 is assumed true and set to 1.

AND-ing combines two values so the result is true (1) if both values are true (that is, non-zero).
OR-ing combines them so the result is true (1) if either value is true.
EXCLUSIVE-OR produces true if one or the other value is true. However, it produces false (0) if both or neither are true.

Before logical operations begin, all operands are converted to 1 (true) or 0 (false). The rule is that everything that is not 0 becomes 1.

BIT-WISE OPERATORS

Bit-wise expressions differ from logical expressions in that there may be many results from a single operation. Bit-wise operations are frequently used to mask I/O bits. For example, the standard SERVO**STAR** SC inputs can be operated on with BAnd (Bit-wise And) to "mask off" some of the inputs. The following example masks off all bits of the digital outputs except the rightmost four:

```
Dim Shared MaskedBits As Long
MaskedBits = System.Dout Band 0xF     'Mask all but rightmost four bits
```

Bit-wise operators only perform bit manipulations on integer operands. Bit-wise operations are always carried out on the MC-BASIC "Long" type—a 32-bit integer. All bit-wise operations produce a 32-bit result from two 32-bit numbers. Bit-wise operations are really 32 independent, bit-by-bit operations. MC-BASIC provides the following Bit-wise operators that are listed in order of precedence:

Bit-Wise Logical Operator	Symbol	Relative Precedence	Unary/Binary	Operand Type
Bit-wise Negation	BNot	1	Unary	long
Bit-wise And	BAnd	2	Binary	long
Bit-wise Or	BOr	3	Binary	long
Bit-wise Xor	BXor	4	Binary	long

Math Functions

MC-BASIC provides a large assortment of mathematical functions. All take either numeric type as input (Double and Long) and all produce Double results. For additional information, refer to the appropriate function description further in this manual.

ABS(X) Return the absolute value of X.
ATAN2(Y, X) Returns the inverse tangent of Y/X in radians.
ATN(X) Returns the inverse tangent of X in radians.
COS(X) Returns the cosine of X.
EXP(X) Returns ex.
LOG(X) Returns the natural logarithm of X.
ROUND(X) Returns the integer nearest to X.
SGN(X) Returns the arithmetic sign of X.
SIN(X) Returns the sine of X.
SQRT(X) Returns the positive square root of X.
TAN(X) Returns the tangent of X.

String Functions

MC-BASIC supports all the common string functions supported in standard BASIC. For additional information, refer to the appropriate function description further in this manual.

ASC(S,I) Returns an ASCII character value from within a string, S, at position, I.
CHR$(X) Returns a one-character string corresponding to a given ASCII value, X.
INSTR(I,SS,S) Returns the position, I, of the starting character of a substring, SS, in a string, S.
LCASE$(S) Returns a copy of the string, S converted to lowercase.
LEFT$(S,X) Returns the specified number of characters from the left-hand side of the string (S).
LEN(S) Returns the length of the string, S.
LTRIM$(S) Returns the right-hand part of a string (S).
MID$(S,I,X) Returns the specified number of characters from the string, starting at the character at position, I.

RIGHT$(S,X)	Returns the specified number of characters from the right-hand side of the string.
RTRIM$(S)	Returns the left-hand part of a string, after removing any blank spaces at the end.
SPACE$(X)	Generates a string consisting of the specified number of blank spaces.
STR$(X)	Returns the string representation of a number.
STRING$(X,{S},{Y})	Creates a new string with the specified number, of characters.
UCASE$(S)	Returns a copy of the string converted to uppercase.
VAL(S)	Returns the real value represented by the characters in the input string.

System Commands

The following commands provide information about the system. You can issue these commands at any time either from the terminal or from the CONFIG.PRG file (with some exceptions). For additional information, refer to the appropriate command description further in this manual.

CAMLIST	This query returns a list of the CAM table names that are defined in the system.
DIR	Lists all the files that exist on the RAM disk and on the Flash disk.
ERRORHISTORY	Displays the log file that contains the last 64 errors that occurred in the system.
ERRORHISTORYCLEAR	Clears the error log file.
EVENTLIST	Lists the names of the existing events and their states.
PLSLIST	Returns a list of the PLS names that are defined in the system.
PROGRAMPASSWORD	This command is used to set the file password and to toggle the password protection state.
RESET TASKS	Reset Tasks removes all tasks from memory, but leaves system variables loaded.
SYSTEM.AVERAGELOAD	This query returns the average realtime load on the CPU.
SYSTEM.CLOCK	Returns the number of system clock ticks.
SYSTEM.CPUTYPE	Returns the type and model of the CPU that was found during the system's power-up.
SYSTEM.DATE	Query or set the date.
SYSTEM.DISKFREESPACE	Returns the amount of free space on the flash disk.
SYSTEM.ERROR	Query the last system error message.
SYSTEM.ERRORNUMBER	Query the number of the last system error.
SYSTEM.ERRORPRINTLEVEL	This controls which errors are printed, according to the severity of the error.
SYSTEM.FLASHDISKSIZE	Returns the size of the flash disk.
SYSTEM.INFORMATION	Returns information that was found during system power-up.
SYSTEM.MAXMEMBLOCK	This query returns the size, in bytes, of the largest block of free memory.
SYSTEM.PEAKLOAD	Returns the peak realtime load on the CPU.
SYSTEM.RAMDRIVEFREESPACE	Returns the amount of free space on the RAM drive.
SYSTEM.RAMSIZE	Returns the size of the RAM that was found during the system's power-up.
SYSTEM.SERCONVERSION	Returns the version number of the SERCON chip.
SYSTEM.SERVICEPRINTLEVEL	Controls (query or set) printing of service messages (ON or OFF).
SYSTEM.TIME	Query or set the current time of day.
TASKLIST	Lists the names and states of loaded tasks.
VARLIST	Returns a list of the variable names that are defined in the system.
VERSION	Returns information pertaining to the version of software loaded into your SERVO**STAR** SC.

Printing

MC-BASIC provides unformatted and formatted printing using the Print and PrintUsing commands.

PRINT AND "?" The Print command (short form, "?") is used to print variables or expressions from the terminal window or from your program.

PRINT # The Print # command is used to print variables or expressions from the HMI interface connected to connector C8.

PRINTUSING PrintUsing (short form, PrintU) allows formatting for printing.

PRINTUSING # PrintUsing # allows formatting for printing from the HMI interface connected to connector C8.

SYNCHRONIZATION

You should be aware that while the Print and PrintUsing commands allow you to print out many variables or expressions on a single line, these variables are not synchronized to each other. In other words, the values can and usually do come from different servo cycles. For example, with the command:

```
Print PCmd - PFb, PE
```

You will find that the values do not correspond, even though PE (Position Error) is, in fact, the difference of PCMD (Position Command) and PFB (Position Feedback). The reason for the discrepancy is that PCMD and PFB come from different servo cycles. They cannot be combined to form PE. If you are monitoring motion and need the command to be synchronized, you must use the RECORD function.

HEX AND DECIMAL MODE

Normally, Longs are printed in decimal format. To change printing to Hexadecimal format, type:

```
System.PrintMode = HEX     'Print numbers in hex format
```

To restore printing to decimal format, enter:

```
System.PrintMode = DECIMAL 'Print numbers in decimal (default) format
```

Printing of double floating-point numbers is not affected by Mode.

Formatting of Longs in PRINTUSING commands is ignored if Mode = Hex.

As an alternative to mode, you can use the keyword "Hex" after the "?" command to print variables and expressions in hex. Subsequent print instructions ("?", PRINT, and PRINTUSING) are not effected.

Flow Control

Flow control is the group of instructions that control the sequence of execution of all other instructions. Additional details on the flow control instructions are contained in this manual. Flow control instructions are:

If…Then…Else…End If
Select…Case...End Select
For..Next
While…End While
While and Sleep
Do...Loop
GoTo

Project

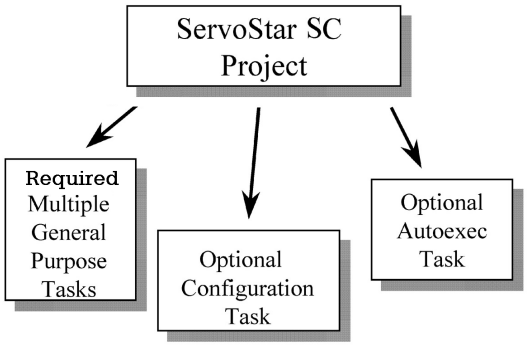
A SERVO**STAR** SC project is the collection of all the software written for an application. The projects in MC-BASIC are multitasking. They can consist of many tasks running concurrently with, and independently of, each other. The project is stored in multiple files, one file for each "task". Tasks are routines running simultaneously with many other tasks. Projects also include other files such as CAM tables and record files. Projects are controlled from either BASIC Moves Development Studio or MotionSuite's Program Development. Each of these applications automatically sets up a separate directory for every project. The following section describes multitasking for the SC.

Project files include tasks and CAM tables. Each project can contain three types of tasks: General-purpose, Configuration, and Autoexec.

General Task All projects contain at least one general purpose task.

Configuration Task An optional configuration task (Config.Prg) automatically declares groups, programmable limit switches (PLS), CAMs, and global variables.

AutoExec Task An optional autoexec task (Autoexec.Prg) starts the application automatically on power-up.



*Structure of a SERVO**STAR** SC Project*

TASKS

A task is a section of code that runs in its own context. Microsoft Windows is a multitasking system. If you open Explorer and Microsoft Word at the same time, they run nearly independently of one another. In this case, both Explorer and Word have their own contexts. They share one computer, but run (more or less), as if the other were not present. Of course, there is inter-task communication. If you double-click on a document in the file manager, it launches Word to edit the file you clicked.

MC-BASIC supports multitasking. You can use tasks to run independent or nearly independent processes on a machine. Also, you can use different tasks to control different operational modes: one for power up, one for set-up, one for normal operation, and another for when problems occur. Like Windows, each task can run independently of the others. Also like Windows, you can set interactions between tasks. The three types of tasks are reviewed: General-purpose, Configuration, and AutoExec.

GENERAL PURPOSE (XXXX.PRG)

General-purpose tasks are the work horse of the SERVO**STAR** SC language. They implement the BASIC logic of your application. The great majority of your programming is executed in general-purpose tasks.

Each general-purpose task is divided into three sections: a task-variable section, a main program, and subroutines. The main program has two subsections at the beginning.

A task-variable definition section
The task-variable definition section is the section where all task variables are declared with the Dim…Shared command.

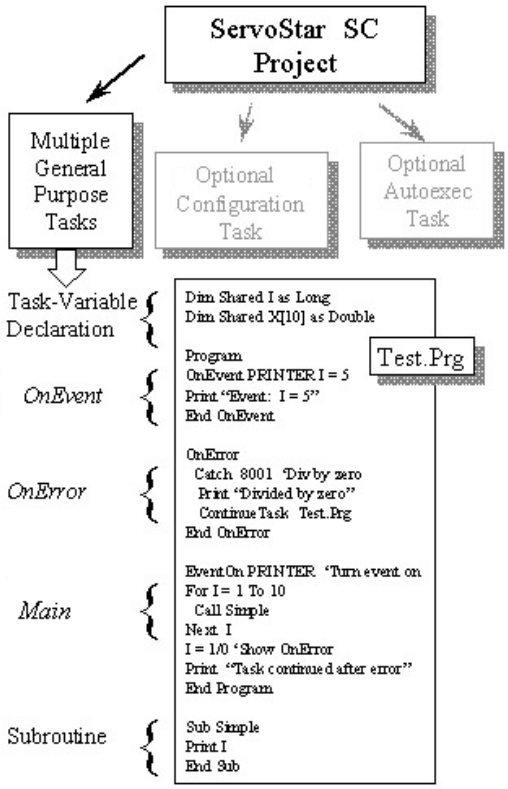
The main program
Most programming is done in the main programming section. The main programming section falls between the Program and End Program commands. The main program itself has two optional sub-sections that must be located after the Program command:

OnEvent sections
OnEvent sections are optional blocks of code that respond to realtime changes such as a motor position changing or an input switch turning on. The main program can contain code to handle (that is, respond automatically to) events. This reduces the effort required to make tasks respond quickly and easily to realtime events.

Event handlers begin with the OnEvent command and end with the End OnEvent command. One OnEvent command is required for each realtime event. Event handlers must be contained wholly within the main program. Event handling is discussed later in this section.

OnError section
The OnError section responds to errors generated by the task. The OnError section allows your program to automatically respond to error conditions and, where possible, gracefully recover and restart the machine. There is at most one OnError sections for each task and it is normally written just after the OnEvent sections.

Optional subroutines
Each task can have any number of subroutines. Subroutines are component parts of the task, and consequently they can be called only from within the task. If you want to call the same subroutine from two tasks, place one copy in each task.

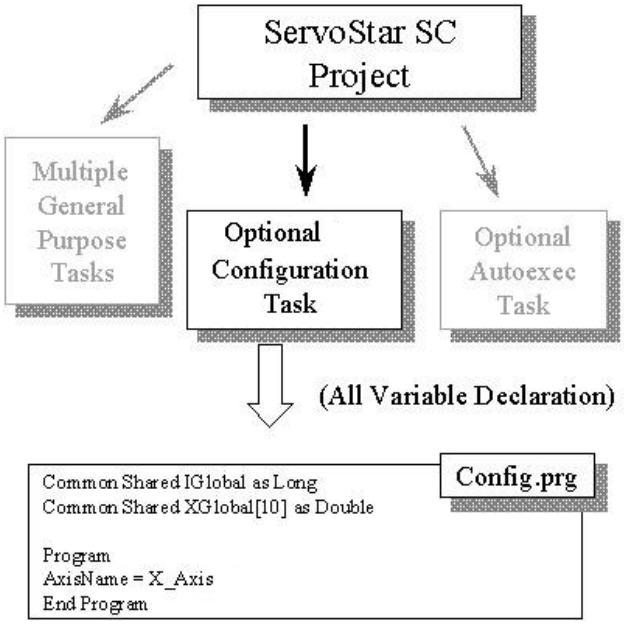


Structure of a Project with Detail on General Purpose Task

CONFIGURATION (CONFIG.PRG)

The optional configuration task is used for declaration of global variables and other constructs such as CAM tables, programmable limit switches (PLS), and group. The key to understanding the configuration task is that all data and constructs shared across tasks must be declared in the configuration task. The name of the configuration task must be Config.Prg.

The configuration task can contain a program. You should rename axes in this program. If you rename an axis anywhere but in the configuration task, only that task has access to the new name.



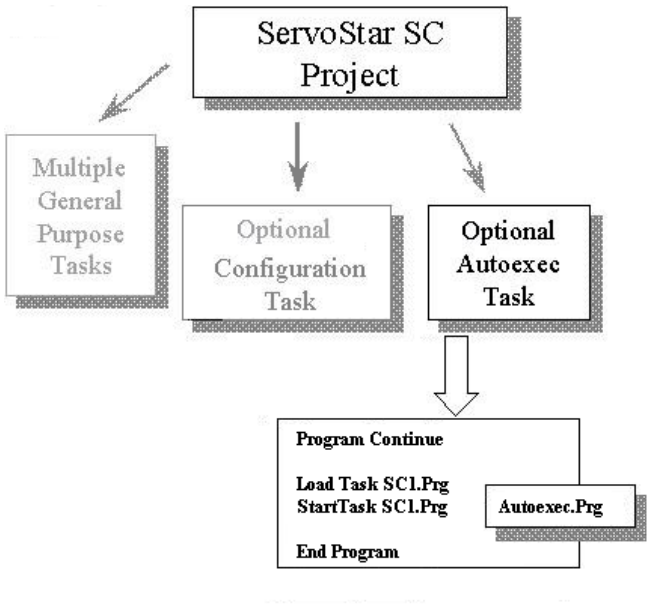
Structure of a Project with Detail on Configuration Task

AUTOEXEC (AUTOEXEC.PRG)

The optional AutoExec task is executed once on power up, just after the Configuration task. Use AutoExec to start power-up logic. For example, you might want to use AutoExec to start a task that sets outputs to the desired starting values. That way, the outputs are set immediately after the SERVO**STAR** SC boots (usually sooner than when the PC boots). The AutoExec task must be called AutoExec.Prg.

The AutoExec task must begin with a program and continue with motion. When starting motion, be sure to command motion from an external controller (digital input) or an operator interface. This ensures that the communication link was established before motion begins.

The AutoExec task should be set to run on power up. To do this, add the keyword "Continue" to the Program command. You should not include OnError or OnEvent sections in the AutoExec. In fact, the AutoExec should be limited to starting other tasks in the system.



Structure of a Project with Detail on AutoExec Task

CREATE AND USE TASKS

When you start a new project, BASIC Moves Development Studio or MotionSuite's Program Development creates the main task as part of opening a new project. After that process is complete, you can add a new task to your project by selecting File, New. You can also press the new task button on the tool bar.

LOADING THE PROGRAM

BMDS automatically loads all tasks in your project when you select Run Project. You can select Run Project by selecting it from the Debug menu, by pressing the F5 key, or by pressing the "Play" button on the tool bar. By default, tasks are loaded from the host PC to the SERVOSTAR SC at a low priority (Priority = 16).

When you select Run Task, the project's main task is started at the lowest priority (Priority = 16). You can change the priority of the main task by selecting View, Project Properties and changing the priority in the bottom of the window. If you structure the software so the main program loads all other tasks, the Run Project button starts your machine running.

PREEMPTIVE MULTITASKING AND PRIORITY LEVELS

Because many tasks share one processor, you must carefully design your system so tasks get processing resources when they need them. You do not want the operator interface taking all the resources when you need fast response to a realtime event. The operating system provides system resources based on two criteria: task priority level and time slice.

PRIORITY LEVEL

Higher priority tasks take all resources until they are completed or waiting for a process to finish.

Avoid Priority=1 since it can disable the terminal.

When you create a program, you must select the task priority level. MC-BASIC allows you to specify 16 levels of priority. The task with the highest priority takes all the system resources it can use. In fact, no task of a lower priority receives any resources until all tasks of higher priority relinquish them. Most systems have one main task running at a medium priority, a background task running at a low priority, and a few high priority tasks. The SC re-evaluates which task has the highest priority at every time slice and assigns resources to the highest priority.

The BMDS Terminal Window relies on the SERVO**STAR** SC command line task that runs at priority 2. If you start a task with priority 1, the terminal is not available until the task is completed or idled. Consequently, you will not be able to communicate with the SERVO**STAR** SC and you may have to power-down the system to recover the terminal window.

As an option, you can set the priority of a task when you start it. For example:

```
StartTask Aux.Prg Priority=6
```

The default priority of events is 1 (highest) and the default priority of programs is 16.

TIME SLICE

Time Slice is a method by which the operating system divides up resources when multiple tasks share the same priority level. The SC provides the first task with one time slice. The next time slice goes to the second, the next to the third, and so on. The time slice is currently one millisecond in duration. This method is sometimes called "round robin" scheduling.

INTER-TASK COMMUNICATIONS AND CONTROL TASKS

Tasks can control one-another. In fact, any task can start, continue, idle, or kill any other task, regardless of which task has the higher priority.

STARTTASK Use STARTTASK to start tasks from the main task. For testing you can also use STARTTASK from the terminal window.

Enter the STARTTASK command in AutoExec.Prg. if you want the task to run on power up. Kollmorgen recommends you not use this feature for tasks that start motion.

IDLETASK This command stops the task at the end of the line currently being executed, and idles all its events. An idled task can be continued (using the CONTINUETASK command) or terminated (using the KILLTASK command). The IDLETASK command does not stop motion that is currently being executed..

CONTINUETASK Tasks that have been idled with the IDLETASK command can be restarted only with the CONTINUETASK command. The command continues an idle task from the point at which it was stopped (idled), or continues task execution after a break point has been reached

KILLTASK This command aborts the execution of a task. The program pointer is left on the line at which the task was stopped, to inform the user at which line the program was killed.

MONITORING TASKS FROM THE TERMINAL

Tasks can be monitored from the terminal using TASK.STATE, TASK.STATUS or TASKLIST.

TASK.STATE TASK.STATE provides the current state of any task. There are two conditions in which the regular task state is modified:
1) when the task is Locked
2) when the task is Interrupted.

TASK.STATUS This query returns the status of a task loaded in memory.

TASKLIST The TASKLIST command returns the state and priority of all tasks that are loaded in the system. You can query TASKLIST only from the terminal window.

RELINQUISHING RESOURCES

When tasks of different priorities compete for processor time, the highest priority task always wins and takes all the resources it needs. However, tasks of high priority can relinquish computer resources under some conditions. In these cases, tasks of lower priority run until the high priority tasks again demand the resources. There are three conditions when a task will relinquish resources: when the task is terminated, when the task is suspended, and when the task is idled.

Terminated Tasks A task terminates when it has finished executing. If a task is started with the NumberOfLoops (NOL) parameter set greater than zero, the task executes the specified number of times and then terminates. At that point, the task relinquishes all resources. Tasks that have terminated remain loaded in the system and can be restarted.

KILLTASK One task can terminate another task by issuing the KILLTASK command. A task relinquishes all resources after the kill command. Tasks that have been killed remain in the system and can be restarted.

Suspended Tasks Tasks relinquish processing resources temporarily when they are suspended. A task is suspended when it is waiting for a resource or is delayed. For example, the SLEEP command suspends a task.

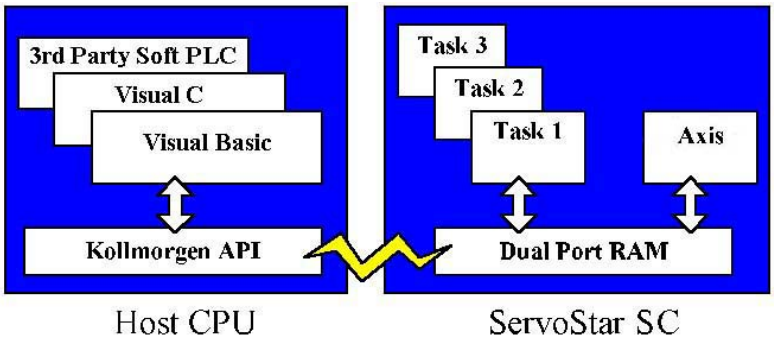
SLEEP Use SLEEP to delay a task for a specific period of time

While and Sleep When using the WHILE command, remember that the CPU repeatedly executes the While block (even if the block is empty). In many cases, this takes all CPU resources. If you want to free up CPU resources during a While block, include the SLEEP command within the block.

Idled Tasks Idled tasks also relinquish resources. This occurs when another tasks issues the IDLETASK command. In this case, resources are relinquished until another task revokes the idle by issuing the CONTINUETASK command.

Multitasking (Advanced Applications)

The SERVO**STAR** SC is a multitasking system in which you can create multiple tasks, which operate independently of one another. Also, because Kollmorgen's API supports multiple applications communicating concurrently, you can write one or more applications, which have access to all tasks and elements as shown in the figure below.



Multitasking is usually used when you have multiple processes largely independent of each other. For example, if you are using the SERVO**STAR** SC to interface to the operator, you will usually use a separate task to execute the interface code.

If a machine is simple to control, you should try to keep the entire program in one task (in addition to the specific function task Config.Prg and Autoexec.Prg, if needed). If you need to use multitasking, you should keep a highly structured architecture. Kollmorgen recommends that you limit use of the main task for axis set up, machine initialization, and controlling other tasks. Normal machine operation should be programmed in other tasks. For example, Initial.Prg might be limited to setting up the axis and then starting Main.Prg, and Operator.Prg. You should not split control of an axis across tasks.

You can use tasks to implement different operational modes.

Multitasking is a powerful tool but it carries a cost. It is easy to make errors that are difficult to find. When multiple tasks are running concurrently, complex interaction can be difficult to understand and difficult to recreate. Limit your use of tasks to situations where they are needed.

Use Tasks for processes that are largely independent. Use event handlers for quick response to realtime events.

Do not create a task as a substitute for an event handler. Events and tasks are not the same. MC-BASIC supports event handlers to respond to realtime events. Events are similar to interrupts in microprocessor systems. They normally run at higher priorities than the programs that contain them. They are ideal for quick responses to realtime events. You normally would not start a new task just to respond to an event. Instead, add the event handler to an existing task.

You also should not use tasks in place of subroutines. Remember that when you start a task, the original task continues to run. When you call a subroutine, you expect the calling program to suspend execution until the subroutine is complete. The behavior of tasks where the two routines continue to execute can cause complex problems.

Knowing when to use multitasking and when to avoid it requires some experience. If you are new to multitasking, you may want to start slow until you are familiar with how it effects program structure.

Event Handler

The main program contains sections that automatically handle events. This reduces the programming effort required to make tasks respond quickly and easily to realtime events. Event handlers begin with OnEvent and end with End OnEvent.

PLACING EVENT HANDLERS

The event handler must be placed in the main program between the Program and End Program commands. Normally, you place OnEvent sections at the top of the main tasks, just after the Program command as OnEvent sections are not loaded until they are encountered.

ONEVENT

After the OnEvent is loaded, you must turn the event on with the EventOn command. Normally, this is done just after the End OnEvent command, as this enables the OnEvent immediately. However, you can enable and disable OnEvent at any time using the EventOn and EventOff commands. Multiple OnEvent can be written sequentially. The SERVO**STAR** SC system supports up to 64 events. The number of OnEvent(s) in a single task is not restricted, so a task may have from 0 to 64 OnEvent(s).

OnEvent does not relinquish resources to the main task until they are complete.

It is important to understand that OnEvent is different from ordinary tasks. OnEvent is preemptive within the task. An OnEvent runs until completion. Then, program execution returns to the main program. While an OnEvent is executing, it does not release CPU time to the parent task or any other task. In this sense, OnEvent is similar to interrupt calls. They run to completion before execution returns to the main program. An OnEvent must have a higher priority than its parent task to ensure that when an event occurs, it interrupts its parent task and runs to completion.

OnEvent is preemptive among other OnEvent. An event of higher priority interrupts an event of lower priority. It is important to note that the rules are valid for a single process (parent task and its events), while events of the respective tasks in the system share the CPU among themselves.

In this example, an event is set up to move the axis to 10000 counts each time the input goes high:

```
OnEvent MOVE_ON_TRIGGER System.Din.1=ON
       Move 10000
End OnEvent
```

Normally, event handlers run at a high priority so that once the event occurs, they run to completion. In most cases, this code should be very short as it usually takes all the resources until it is complete.

SCANTIME is in cycles of the SERVO**STAR** SC update rate. This is normally 1 milliseconds. So, setting SCANTIME to 5 configures the system to check the event condition every 5 milliseconds. For example:

```
OnEvent System.Din.2 = ON ScanTime = 5
```

Even events that are not executing take processing resources. This is because the condition must be checked regularly. You can reduce the resource load by keeping the conditions simple and setting the scan time to as long a time as is practical.

Be cautious when setting the priority of events. When the priority of events is too high, they take excessive resources. You generally would not set the priority of an event that watches a push button to priority 1 because the job is not so important that it should take all the resources. On the other hand, events should be at least as high as most of the general tasks. Otherwise, the event might fire but not execute. Remember that all tasks of a certain priority must relinquish resources before any tasks of a lower priority execute. Under no circumstances should the priority of an OnEvent command be equal to or lower than the priority of the task wherein it runs.

CONTROLLING ONEVENT OCCURRENCES

Events can be controlled from within the task in which they reside or from the terminal. The main program or any subroutine can issue EventOn (enable) or EventOff (disable). OnEvent cannot be controlled from other tasks.

EventOn EventOn enables the OnEvent Command. The EventOn command must come after the definition of the OnEvent.

EventOff EventOff disables the OnEvent Command.

EventList EventList provides a complete list of all events in all tasks of all events with their name, the task name, the priority, whether the event is armed, and current status.

EventList can be issued only from the terminal window.

EVENTS AT START-UP

Events are normally triggered by the OnEvent condition changing from false to true. A single transition in the condition is required to run the ONEVENT command. One exception to this is start-up. At start-up, if the condition is true, the ONEVENT command executes once, even if there has not been a transition.

PROGRAM FLOW AND ONEVENT

You can use GOTO commands within an OnEven block of code. However, because OnEvent interrupts the main program, you cannot use GOTO to branch out of or into the event handler. You cannot place OnEvent…End OnEvent in the middle of program flow commands (For…Next, If…Then, etc.). Also, you cannot declare or use local variables inside an OnEvent block.

Error Handler

Error handlers let you write your program to respond to errors in much the same way as event handlers let your program respond to events. There are three contexts in which an error can occur and be handled: task, system, and terminal. You cannot handle errors that occur in the terminal context.

Error handling refers to the methods used to react to and process fatal and non-fatal errors that can occur during the operation of the SERVO**STAR** SC. Fatal faults cause a watchdog timer to be triggered. All errors have corresponding default actions to be taken when they occur. The following definitions will help you:

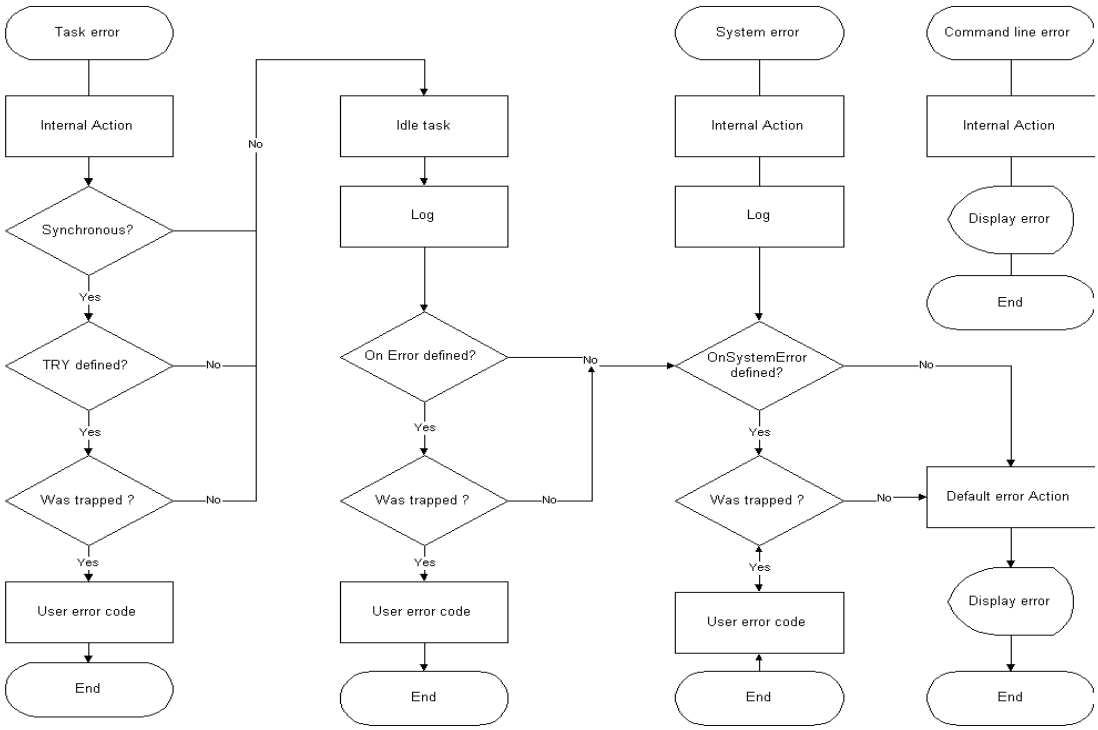
Term	Definition
Internal error action	Immediate action taken by the system software when an error occurs. This action cannot be turned off or bypassed by the user.
Default System error action	Action taken by the system according to the context and severity of the error if the error is not handled by the user. All default system error actions can be bypassed by the user by defining error handler functions.
Synchronous error	Error caused by the user task and detected by the interpreter. This type of error is associated with a specific line of program code in the user defined task. Examples include division by zero and out of range parameters in a Move command. The task that caused the error is stopped.
Asynchronous error	An error caused by the user task that is not associated with a specific line of program code. Examples include following error and motion overspeed.
System error	Error not associated with user tasks or command lines.

When an error occurs in the SERVO**STAR** SC system it can occur in one of three contexts: task, system or terminal. It is important to recognize the difference between these three since the action taken by the system varies depending on the context. Each of these is described in detail below.

TASK ERRORS

Errors occurring as a result of an executing task take place in the task context. Asynchronous and synchronous errors can occur in this context. A specific line of code generates synchronous errors. An asynchronous error that occurs in a task is not associated with a particular line of program code. A good example of an asynchronous error caused by a task is a Following Error, which occurs due to a difference between commanded position and actual position. This error may occur when an element being commanded to move comes up against a mechanical obstacle.

The following flow chart shows the error processing flow that occurs in the Task context.



During the Internal error action the synchronous error is logged in the SERVO**STAR** SC and the task is idled. The default system error handler stops all motion and all attached tasks. In the above flowchart, there are two mechanisms for trapping and dealing with errors: OnError and Try/Finally blocks. These two error handling mechanisms allow you to write your program to respond to errors.

ONERROR

The OnError block is handled similarly to an event handler. An event handler lets your program respond to events and the OnError block allows your program to catch errors. OnError overrides default system error action if error is trapped by OnError. Main program execution is stopped while the OnError block is executing and must be resumed if that is desired. An example of this is shown later in this section. To add an OnError block to your program add an OnError code block between the Program and End Program section of your program. The syntax of OnError is:

```
Program                                    ' Beginning of program
Program                                    ' Beginning of program
<program code>
       OnError                              ' Start of OnError block
              catch <Error Number X>
                     {code to execute when error X occurs}
              catch <Error Number Y>
                     {code to execute when error Y occurs}
              catch else
                     {code to catch all other errors}
       End OnError                                ' End of OnError block
<program code>
End Program                               ' End of program
```

The number of catch statements in an OnError block is not explicitly limited. The CATCH else statement may optionally be used. If you want a task to resume after an error has occurred, you should use the CONTINUETASK command (see the example in Appendix C).

You can use the GOTO command within the error handler. However, because OnError interrupts the main program, you cannot use GOTO to branch outside the error handler. You can not place an OnError block in the middle of program flowThe scope of OnError is that of the task that it is contained within. It handlers errors that occur within that task only (see the example in Appendix C).

TRY/FINALLY

Unlike an OnError block, the Try/Finally block may appear anywhere in the main program section of the task. It can be used to take specific action with relation to a particular area of your program code. This type of error handler block can only trap synchronous errors in the task context. The block is started by the Try keyword and is terminated by the End Try keyword. The syntax for the Try/Finally block is:

```
Program                          ' Beginning of program
       <program code>
       Try                         ' Start of Try block
              <code being monitored>
                     catch <Error Number X>
                            {code to execute when error X occurs}
                     catch <Error Number Y>
                            {code to execute when error Y occurs}
                     catch else
                            {code to catch all other errors}}
                     Finally
                            {executed when error is trapped}
       End Try                     'End Try block
       <program code>
End Program                      'End of program
```

Try blocks can be nested. The program lines between the line causing the error and the matching catch statement are skipped. The interpreter will try to trap an error starting with the innermost catch statement. If there is no matching catch statement, the error is handled as a regular synchronous error. Errors trapped inside the Try-block are not logged. The Finally statement is executed only if an error occurred and was caught inside the Try block.

SYSTEM ERRORS

The system context is the lowest level of the three contexts. It refers to errors not directly related to specific tasks. Errors that occur in this context effect all tasks that are running. The default system error handler processes these errors. Examples of system context errors include floating-point unit errors, CPU errors, and errors that occur on motion elements not attached to specific tasks.

ONSYSTEMERROR

OnSystemError is used to trap and process all errors in the system context. It is the upper level of the heirarchial error processing structure formed by the combination of Try, OnError and OnSystemError. OnSystemError may be written in the body of any task, but only one instance may exist in the system at any time. It is used to trap both synchronous and asynchronous errors in all tasks, as well as errors that occur within the context of the system.

A system error is one that is not associated with a specific task. An example of a system error is a position following error that occurs due to some external force being applied to an axis that is not attached to a task. When an error is trapped, the specified error processing code is run and the task is stopped. The task will be in state 4. It is possible to continue task execution by explicitly entering the CONTINUETASK command within the error processing code, but the task will continue only after the error has been corrected. OnSystemError may be used to trap errors that are not specifically trapped by Try or by OnError. It is then used, for example, to execute an orderly shutdown of the system, or an orderly recovery procedure.

The following is the structure of the OnSystemError block:

```
Program                                 'Beginning of program

      <Code>
      OnSystemError                        'Start of OnSystemError block
            -catch <error number X>
                  <code to execute when error X occurs>-
            -catch <error number Y>
                  <code to execute when error Y occurs>-
            -else
                  <code to execute for all other errors>-
      End OnSystemError                     'End of OnSystemError block
      <Code>

End Program                              'End of program
```

ERROR PRINT LEVEL

The command SYSTEM.ERRORPRINTLEVEL allows you to control which types of system errors are printed to the message log window. There are three levels of system errors; fatal faults, errors, and notes.

The syntax for the command is:

```
Sys.ErrorPrintLevel = <value>
?Sys.ErrorPrintLevel
```

Where <*value*> specifies the level(s) you want to print to the message log:

0 (SILENTLEVEL)	Fatal faults, errors, and notes are not printed.
1 (FAULTLEVEL)	Fatal faults are printed, notes and errors are not printed.
2 (ERRORLEVEL)	Fatal faults and errors are printed, notes are not printed.
3 (NOTELEVEL)	Fatal faults, errors, and notes are printed.

This command affects only printing to the message log window. Fatal faults and errors continue to be logged, and can be viewed using the command ?ErrorHistory. The ErrorPrintLevel command applies only to asynchronous errors; synchronous errors are not effected.

TERMINAL ERRORS

The terminal context is similar to the task context previously described, as commands are processed in a similar manner. The error action is different from a task as there is no interpreter to be idled when an error occurs.

Software WatchDog

The SERVO**STAR** SC provides a watchdog timer to monitor periodic tasks. By default, WatchDog monitors only system tasks like the profile generator. WatchDog also monitors user tasks, but the task must trigger the WatchDog at least once per specified number of cycles.

WATCHDOG COMMANDS

The following commands are used to configure the SERVO**STAR** SC to expect watchdog cycles:

WDINIT The WDINIT command is used to establish a WatchDog timer

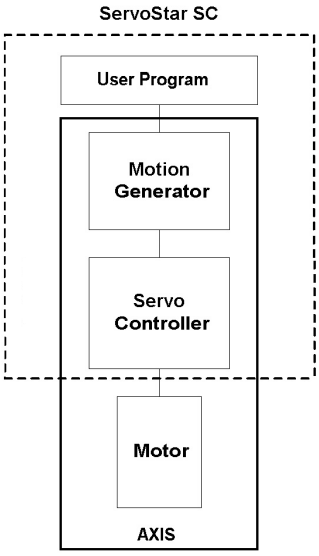
WDCYCLE The WDCYCLE command is used to cycle (trigger) the WatchDog timer.

WDDelete The WDDELETE command is used to delete the WatchDog timer.

This page intentionally left blank.

SETTING UP AN AXIS

The SERVO**STAR** SC controller is oriented around an axis. An axis is a combination of a motor, and some portions of the SERVO**STAR** SC. A diagram of an axis is shown below.



*SERVO**STAR** SC Axis*

The axis has a set of properties defined by the SERVO**STAR** SC. These properties are used in the calculation of position and velocity commands, monitoring limits, and stores configuration data for the axis. All these components together form an axis of motion.

Starting Position

Before we get started here, you will need:

Well-tuned control loop
You should have followed the process for installing and tuning the control parameters according to the *SERVO**STAR**® SC Installation Manual*. You should have run **MOTIONLINK** to select your motor type and tuned the axis to your satisfaction.

Install and check out the SERVO**STAR SC**
You should have installed and checked out your SERVO**STAR** SC according to the *SERVO**STAR**® SC Installation Manual*. You should have wired all I/O and followed the procedures in the *SERVO**STAR**® SC Installation Manual*.

Properties set through the terminal mode can be saved for the next power up after issuing the STORE command.

Install BASIC Moves Development Studio or MotionSuite
You should have installed either BASIC Moves Development Studio or MotionSuite. Remember that any commands typed in through the terminal window are lost when you power down. These commands need to be entered in your SERVO**STAR** SC program to be recalled at power up.

Axis Setup Subroutine

A common practice is to take all the axis setup parameters or properties and place them in a subroutine. The subroutine is called early in your main task. A sample program with an axis setup subroutine is found in Appendix C. The program has these sections.

- A short task-variable declaration
- An example program with:
  - Intro section that calls the axis setup soubroutine and then enables the drive
  - A section that generates simple motion
- An axis setup subroutine

Acceleration and Deceleration

Two axis properties (ACCELERATION and DECELERATION) control the acceleration rates. The following lines of code entered at the terminal window (or in your program) changes the acceleration rates for the axis:

```
Acceleration = 1000
Deceleration = 1000
```

Or you can use the short form:

```
Acc = 1000
Dec = 1000
```

S-Curves

Jerk is the first derivative of ACCELERATION. Fast moves usually generate large amounts of jerk. Having a large amount of jerk in a motion profile can excite machine resonance frequencies and thereby cause unnecessary wear and noise. Some motion controllers simplify motion profiles by instantaneously changing the acceleration command that implies a very high level of jerk. The SERVO**STAR** SC allows you to limit the amount of jerk in all profiles by using the axis properties SMOOTHFACTOR or JERKFACTOR.

SmoothFactor

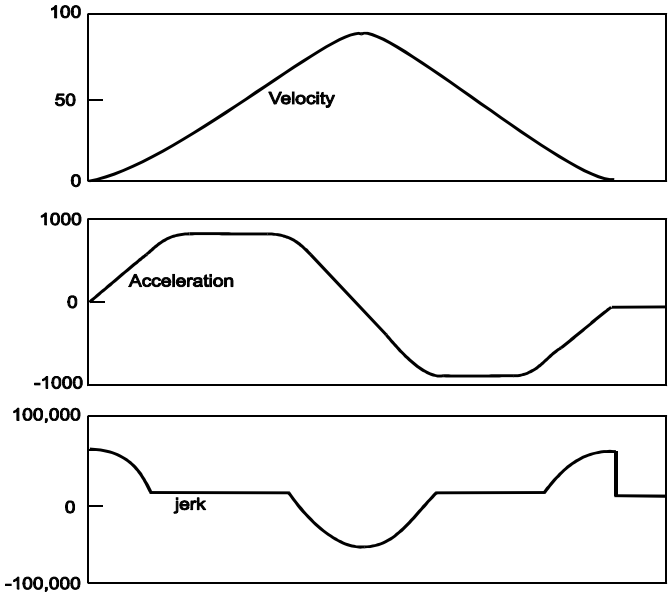
You can specify trapezoidal profiles by setting SMOOTHFACTOR (Short form: SMOOTH) to zero. If you want a smoother acceleration, you can increase the SMOOTHFACTOR from 1to a maximum of 100. If SMOOTHFACTOR is large (greater than 50), it can increase the acceleration time by one or more orders of magnitude.

```
SmoothFactor = 0      'Trapezoid
Smooth = 10           'Some smoothing
Smooth = 100          'Maximum smoothing
```

Acceleration Profile

The SERVO**STAR** SC provides smooth, controlled acceleration profiles to reduce vibration and wear on the machine, and to allow high acceleration rates. The SERVO**STAR** SC allows you to independently control acceleration and deceleration to further reduce vibration.

Using ACCELERATION and JERK to limit velocity changes produces acceleration profiles that remove the high frequency components of torque that are present in straight-line acceleration. This minimizes the excitation of machine resonance frequencies, which in turn, reduces audible noise, vibration, and mechanical wear. For example, the figure below shows a typical SERVO**STAR** SC acceleration profile showing how controlled jerk produces smooth acceleration:



Acceleration Profile

Position and Velocity

Position and velocity are the key command and feedback signals for each axis. All these properties are double-precision floating-point numbers in count units. The four properties that hold these values are:

- POSITIONCOMMAND or PCMD
- POSITIONFEEDBACK or PFB
- VELOCITYCOMMAND or VCMD
- VELOCITYFEEDBACK or VFB

For example:

```
? PFb              'Read feedback
X1 = VFb           'Set X1 to the axis current velocity
```

Rotary Mode

The SERVO**STAR** SC allows you to set up the axis in your system as a rotary. A rotary axis is often used for mechanical mechanisms that repeat after a fixed amount of rotation. For example, a motor driving a rotating table is often configured as a rotary. In this case, the table units may be set up as degrees and the axis can be set to repeat after 360 degrees. In that way, the position repeats after every rotation of the table rather than continuing to increase indefinitely. The key to the rotary mode is setting rollover position (POSITIONROLLOVER or PROLLOVER) correctly and accurately (see Appendix C for an example).

Rotary Mode Accuracy

When using rotary mode in applications where the motor turns continuously in one direction, it is important to take advantage of all available accuracy in the SERVO**STAR** SC. This is because inaccuracy is accumulated over the many revolutions of the motor.

Limits

Limits imposed on the SERVO**STAR** SC system help protect the machine from excessive excursions of travel, speed, and torque. There are three types of limits in the SERVO**STAR** SC system:

- Generator Limits
- Realtime Limits
- Drive Limits

Generator Limits

Generator limits affect subsequent commands to the motion generator. For example, if you change an acceleration limit, this has no effect on the current motion command, but it applies to subsequent motion commands. If the axis is not in a Jog, analog mode, or acting as a slave, then POSITIONMIN and POSITIONMAX (position limits) are checked only at the beginning of the move.

Realtime Limits

Changes in these limits affect current operations. VELOCITYFEEDBACK is checked against VELOCITYOVERSPEED in each update cycle.

Drive Limits

Changes in drive limits impact current operations. For example, ILIM limits the peak current that can be output by the SERVOSTAR SC. These limits are imposed independently of the position and velocity commands issued by the controller.

Position Limits

There are several limits in the SERVOSTAR SC related to position:

- PMIN
- PMAX

PMin and PMax

PMIN and PMAX place minimum and maximum limits on the position feedback for an axis. You can set the limits with these commands:

```
PositionMin = -10
PositionMax = 200.5
```

Or you can use the short form:

```
PMin = -10
PMax = 200.5
```

Position limits must be enabled to operate.

Position limits must be enabled before they can operate. Limits are enabled with POSITIONMINENABLE and POSITIONMAXENABLE.

Position limits are checked two ways, in the motion generator and each update cycle. If you command a position move beyond the position limits, an error is generated and the MOVE does not start. If the axis is moving and crosses the position limit, this generates an error. If the axis has already been stopped because the position limits were crossed, you can enter commands that move the axis back within the limits. The axis moves without generating errors.

Position limits are not continuously checked unless the SERVO**STAR** SC is in master/slave mode. There are conditions that occasionally result in the motor moving outside PMIN or PMAX that are not detected by the SERVO**STAR** SC.

POSITION ERROR

Position error is defined as the difference between the position commanded and position feedback. When an axis is at rest (settled), this simple definition of position error is valid. However, when the axis is in motion, the true position error does not equal the instantaneous commanded position minus the instantaneous feedback position because there is a time delay of one cycle time (1 ms) from when the move command is issued until the feedback position is received. When the SERVO**STAR** SC calculates position error, it automatically accounts for the communications delay.

PositionMinEnable/PositionMaxEnable

You can control whether the SERVO**STAR** SC watches either one or both of the limits in PositionLimit. This is done by setting POSITIONMINENABLE and POSITIONMAXENABLE to either ON or OFF:

```
PositionMinEnable = ON
PositionMaxEnable = ON
```

Or, you can use the short forms:

```
PMinEn = ON
PMaxEn = ON
```

You must turn on PMINEN and PMAXEN for PMIN and PMAX to function.

There are also the hardware position limits (i.e., the limit-switches). They are enabled by setting LIMDIS to 0 and the appropriate value of inXmode (X = 1,2,3).

MAXIMUM POSITION ERROR

You should limit the maximum position error your system will tolerate. You do this by setting the axis property POSITIONERRORMAX (PEMAX).

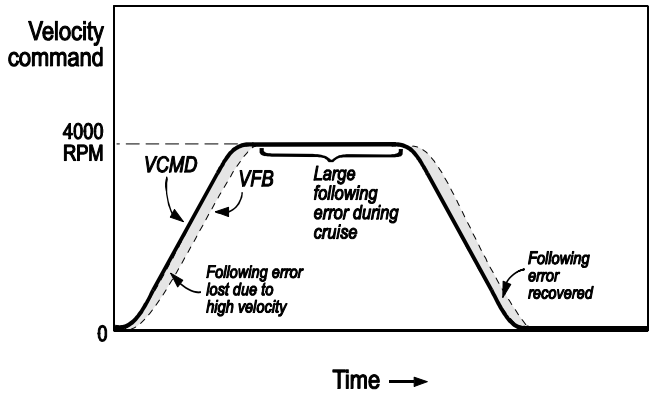
Set PEMAX to a value that matches the needs of the application. When the actual position following error (PE) exceeds PEMAX, the motion is stopped. If the motion is already stopped when this condition is detected, the axis is disabled.

During normal operation, occasional occurrences of position error overflow usually indicate a malfunction of the machine, such as a worn or broken part, or a need for lubrication. Set the maximum position error well outside the boundaries of normal machine operation or nuisance errors occur when the machine has been running a while.

During installation, position error overflow frequently occurs because the system is unstable. In this case, the motor runs away even though zero velocity is commanded. Set POSITIONERRORMAX to a reasonably small value before powering the system up. For example, you might set it to a few revolutions of the motor or a few inches or centimeters (for linear systems). This way, if the tuning is unstable, the system is less likely to cause a problem. Setting PEMAX to a very large number prevents the position error overflow error from detecting an unstable system and the motor is able to run away.

POSITION ERROR WITH PROPORTIONAL POSITION LOOP

The proportional position loop is the simplest position loop. The velocity command is proportional to the following error. Large velocity commands need large following error limits. The constant of proportionality is GP (position loop gain). At first, this may seem confusing. Actually, it just means that for each count of position error, a command of GP RPM is given. Typical values of GP range from 1 to 5. If GP were say, 2, and the commanded speed were, say 200 RPM, there would necessarily be 1 of following error. The following error for acceleration with a typical proportional loop is shown below.

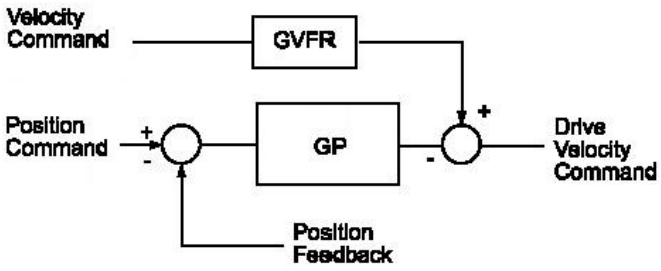


Following Error with Proportional Position Loop

REDUCING POSITION ERROR

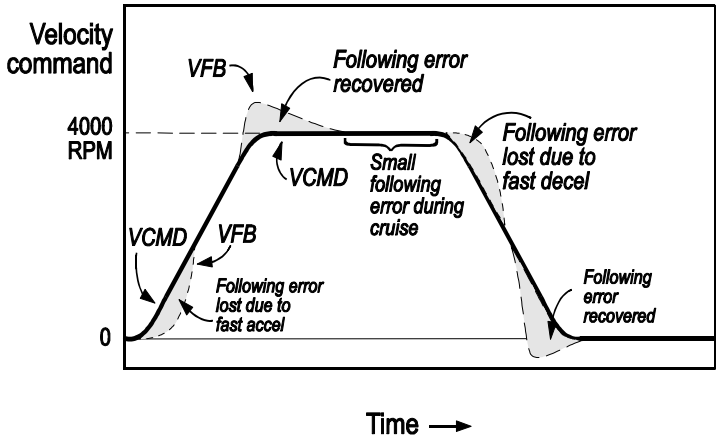
Position error generated during a move is often called "following error" because it measures how well the motor follows the command. Reducing the following error is equivalent to following the command more closely, which normally is desirable. Commonly, you will set GP to be large, which reduces following error because the higher gain means it takes less position error to generate the same velocity command. There is a limit though, because if GP gets too large, the position loop becomes unstable. After GP is as large as possible, what else can you do to reduce position error?

One technique to reduce following error is to add a feed-forward term to the velocity loop. This allows you to generate velocity commands with a combination of the profile velocity command and position error, rather than relying wholly on position error. A position loop with feed-forward is shown in the following figure.



*SERVO**STAR** Proportional Loop with Feed-Forward*

In this case, you can reduce following error by raising the gain GVFR. When GVFR is at full scale (100%), the following error at steady-state is reduced to zero. The only following error is generated during transients. The effect of 100% feed-forward on following error is shown in the following figure.



Following Error with Proportional Loop and Feed-Forward

The benefit of 100% feed-forward is that it eliminates steady-state following error. The drawback is that the system overshoots when subjected to acceleration or deceleration. In some cases, this is not an issue because the system may always transition smoothly, or some overshoot may be acceptable. However, in many cases 100% feed-forward is not acceptable. In these cases, you can reduce the feed-forward term to reduce overshoot. The larger the feed-forward gain, the greater reduction you will see in steady-state following error. Most systems can tolerate the overshoot generated by feed-forward gains of 50%.

POSITIONERRORSETTLE

The axis property POSITIONERRORSETTLE defines what level of position error is considered close enough to zero to be settled. For example, you can use:

```
PositionErrorSettle = 1
```

Or you can use the short form:

```
PESettle = 1
```

To specify that the position error must be below1 count to be considered “settled.”

POSITIONERRORSETTLE is used when the motion controller must determine when an axis is close enough to the final end point of a move to be considered “settled.” This is commonly used between moves to ensure that the response to the first move is complete before moving to the second. The SERVO**STAR** SC does this automatically through STARTTYPE when executing MOVE commands (“Single Axis Motion”).

You must set TIMESETTLE (TSETTLE) > 0 or POSITIONERRORSETTLE is ignored.

Acceleration Feed-Forward

Acceleration feed-forward allows the use of 100% velocity feed-forward with no overshoot. Acceleration feed-forward works by adding current equivalent to the torque required to accelerate a fixed load. Acceleration feed-forward effectively cancels the torque generated by the inertia changing speed. This is why sometimes this technique is called inertial feed-forward.

Acceleration feed-forward works only when the inertia load is proportional to the axis acceleration. It does not work with an axis that is coupled (non-rectangular robots). It also does not work well when the axis inertia varies. However, for most applications, acceleration feed-forward reduces overshoot and following-error significantly.

To use acceleration feed-forward, you can use **MOTIONLINK**, or you can set the acceleration feed-forward scaling constant from the SERVO**STAR** SC. The correct line of code is:

```
GPAFR = 1000
```

The valid range for this value is 0 to 2000.

The purpose of the acceleration feed-forward is to zero the following error during constant acceleration. The automatic design (user gain = 1000) should give good results. The user may fine tune this value by applying trapezoidal trajectory, and find the user gain that yields the minimum following error during the acceleration and deceleration.

TimeSettle

TIMESETTLE (or TSETTLE) defines the amount of time the position error must be lower than the value of PESETTLE before the move is considered "settled." After the move profile is complete and the position error is less than POSTIONERRORSETTLE, the SERVO**STAR** SC waits TIMESETTLE milliseconds for settling. If the position error goes above POSITIONERRORSETTLE, during that time, the timer is reset.

IsSettled

ISSETTLED is a logical (TRUE or FALSE) property that indicates if the axis is settled. To be settled, the motion profile must be complete and the position error must be below POSITIONERRORSETTLE for a period of TIMESETTLE milliseconds.

IsMoving

ISMOVING is a property that indicates the state of the motion profiler. The valid range of values is from 2 to 3, with the following meaning:

-3	Analog Mode
-2	Procedural command (HOME, TUNE, ENCSTART, STEP, ….)
-1	Element is a slave (gear or CAM) unless an incremental move is issued, in which instance the following values are valid:
0	Element is not moving
1	Element is accelerating
2	Element is at constant velocity phase (cruise)
3	element is decelerating

Example:

```
While ismoving  > 0                'wait for profiler to finish
End While
```

Axis Velocity Limits

There are several limits in the SERVO**STAR** SC related to velocity:

VELOCITYOVERSPEED or VOSPD
VELOCITYLIM or VLIM
ACCELMAX
DECELMAX

VelocityOverspeed

VELOCITY OVERSPEED is a property that sets the maximum speed the axis can travel. VEOCITYOVERSPEED is in SERVO**STAR** SC velocity units and is set at any time. It is checked every millisecond. You can set VELOCITYOVERSPEED with the following command:

```
VelocityOverspeed  = 4000
```

VelocityLim

VELOCITYLIM (VLIM) is a property that sets the maximum speed that the motion generator can command. VLIM is in SERVO**STAR** SC velocity units and can be set only from the terminal window or in Config.prg. It is checked at the beginning of each move.

In the system, several different velocity variables exist. Their relationships are defined as:

$$\begin{matrix} vel_{cmd} \\ vel_{fbk} \end{matrix} \leq vel_{cruise} \leq VLIM \leq VMAX \leq \begin{cases} MSPEED \\ (VBUS * 0.707 / MBEMF) * 1000 \\ 24000 \\ 180000000 / MENCRES \\ 192000 / MRESPOLES \end{cases}$$

You can set VLIM with the following command:

```
VelocityLim = 5000                  'Velocity Limit on Drive
```

Or use the Short form:

```
VLim = 5000                  'Velocity Limit on Drive
```

Axis Acceleration/Deceleration Limits

Two limits control acceleration and deceleration that limit the velocity transients on acceleration profiles. These properties may be set from the terminal or in a user task (or any other context).

ACCELMAX

ACCELMAX is the upper limit for acceleration in motion commands. It is in SERVO**STAR** SC acceleration units. ACCELMAX is set up in the BMDS auto-setup program run when you start a new project.

DECELMAX

DECELMAX is the upper limit for deceleration in motion commands. It is in SERVOSTAR SC acceleration units. DECELMAX is set up in the BMDS auto-setup program run when you start a new project.

Check Your Settings

You can verify your settings by typing them in from the terminal window and using the MOVE command to execute motion First make sure your drive is compensated for the correct motor and tuned for the machine load. Enable your SERVO**STAR** SC. Next, Set the Conmode =2, and software enable the drive.

To get motion from the SERVOSTAR SC, be sure you have enabled both the 24V input on C3 and the motion connector on C9. Otherwise, there will be no motion.

Motor Compensation

Most SERVO**STAR** SCs are shipped from the factory already configured for a particular motor. Verify this by applying logic power and monitoring the Status Display. A factory-configured SERVO**STAR** SC goes through a power-up sequence flashing all segments of the display, then it flashes a "0" for a few seconds, then it flashes a - "S".

SYS.CONMODE

In order to enable the SERVO**STAR** SC, you must have enabled both the 24V input on C3 and the motion connector on C9. Additionally, you must enter:

```
SYS.MOTION=1
SYS.CONMODE=2
```

If the SERVO**STAR** SC is not configured with a particular motor, the display will flash a minus one (-1), indicating that you must enter SERVO**STAR**, motor, and application variable parameters.

Take care when applying power to the SERVOSTAR SC. It is factory configured to enable upon application of power. Verify that the hardware Remote Enable (REMOTE) enable switch is disabled.

Enabling

Enable the SERVO**STAR SC** with the following command:

```
Enable = ON
```

Or use the Short form:

```
En = ON
```

The SERVO**STAR** SC is enabled. This condition is indicated with a "0" or "8" on the front of the SERVO**STAR** SC. A "decimal point" to the bottom-left of the "0" or "8" on the SERVO**STAR** SC comes on when it is enabled. The decimal point may flash under certain conditions.

You must turn on SYS.ENABLE before turning on any of the drive enable properties because when the SYS.ENABLE is off, it forces all the axis enable properties off.

Motion Flags

The next step in preparing a system for motion is turning on the motion flag. Enter the following command to turn on the motion flag:

```
Motion = ON
```

Now, you can perform a MOVE command. For example, enter:

```
Move 1000 VelocityCruise = 10
```

To move to position 1000 with a velocity of 10. You should see the motor move.

Use a hand tachometer or other speed-sensing device to verify that the speed settings are correct. Move to various positions using the incremental move (by setting Absolute to OFF):

```
Move 1000 VelocityCruise = 10 Absolute = Off
```

Loop Types

The SERVO**STAR** SC system supports three main control loop types:

- Position Control with standard feedback (motor sensor) through connector C3. This is OPMODE 8.
- Position Control with Dual-feedback (OPMODE 8, DUALFB = 1)
  - Velocity Loop Feedback and motor Commutation Control feedback from standard feedback (motor sensor) through connector C3.
  - Position Loop Feedback from external feedback (load sensor) through connector C5
- Velocity Control with standard feedback (motor sensor) through connector C3. This is OPMODE 0.

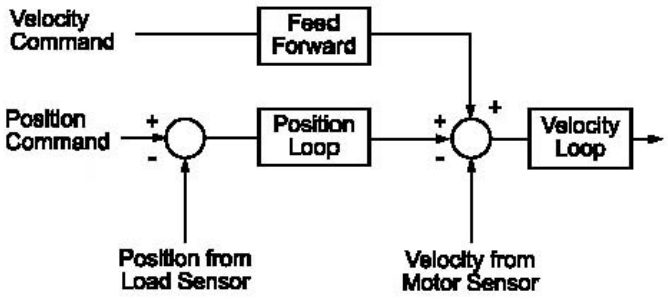
The SERVO**STAR** SC sends position and velocity commands each update cycle. The operation of the SERVO**STAR** SC is hardly effected by the loop selection. The loop selection is made in the SERVO**STAR** SC.

Position Loop

Position loop is the standard operating mode of the SERVO**STAR** SC system. Position Loop is available in the digital position mode only (OPMODE 8). For position operation, set the position loop gain to any value greater than zero. The normal way to set up the position loop is by using **MOTIONLINK**. See the *SERVO**STAR**® SC Installation Manual* for more information.

Dual Loop

Dual Loop is available in the digital position mode only (OPMODE 8). Dual loop is used to compensate for non-linearity in mechanical systems (inaccuracy from gears, belt drives, or lead screws, or backlash from gears and lead screws). Dual loop works by adding a second feedback sensor (usually a linear encoder), to the system so that the load position can be determined directly. Unfortunately, the load sensor cannot be the sole feedback sensor in most systems. This is because the mechanical connection between the motor and load is usually too compliant. Reliance wholly on the load sensor usually causes machine resonance, which can severely effect system response. Dual loop forms a compromise: use the load sensor for position feedback where accuracy is critical, and use the motor sensor for velocity feedback where low compliance is required. This forms the dual loop shown in the following figure.



Dual Loop Control

The additional hardware for dual loop is a second encoder connected to the SERVO**STAR** SC running dual loop. You configure the SERVO**STAR** SC for dual loop operation using **MOTIONLINK**. See the *SERVO**STAR®** SC Installation Manual* for more information.

The SERVO**STAR** SC is not directly involved in dual loop operation. In a sense, the SERVO**STAR** SC is not aware that the drive is operating in dual loop. However, when setting up the feedback system, you must configure your units for the external encoder, not the motor feedback device. In addition you must also tell the SERVO**STAR** SC to monitor the external encoder for position error (XENCRES, XENCDIR). This is done with the axis feedback property. For example:

```
DUALFB = 1                    'Set the axis feedback to the load
```

There are two velocity modes available in the SERVO**STAR** SC. These are analog and digital. The velocity mode is set by issuing the opmode dvmode command. See OPMODE, ANALOGVELOCITSCALE, and ANIN1 for further details.

SINGLE-AXIS MOTION

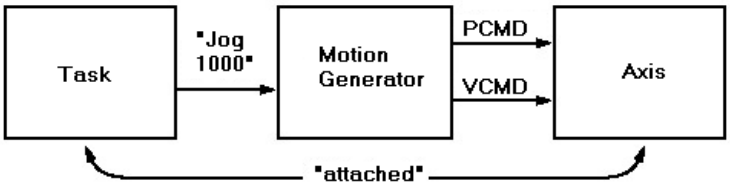
The SERVO**STAR** SC supports two main types of motion:

- Single-axis motion
- Master-Slave motion

A "motion generator" controls all motion in the SERVO**STAR** SC. This software device receives commands from tasks and produces position and velocity commands every update cycle. The two main single-axis motion commands are JOG and MOVE. The SERVO**STAR** SC provides motion command execution synchronization as well as on-the-fly motion profile changes. In addition, the SERVO**STAR** SC provides multiple modes for starting and stopping motion. All motion commands are updated every cycle time (1ms).

Motion Generator

The basis of all motion in the SERVO**STAR** SC is the "motion generator." When you issue a motion command such as JOG or MOVE to an axis, a motion generator is created in software. That motion generator controls the position and velocity commands of the axis specified in the MOVE command. At regular intervals (normally 1 ms), the motion generator updates these commands. At the end of motion, the motion generator is terminated. The figure below shows the normal operation of a motion generator.



Conditions for Motion

Several conditions must be met before the SERVO**STAR** SC will generate motion:

- The controlling task must "attach" the axis.
- The SERVO**STAR** SC must be in SYS.CONMODE = 2
- The SERVO**STAR** SC must be software and hardware enabled.
- The system motion flag must be enabled.

Attaching Motion to the Task

Before a task can command motion, it must be attached to the axis. This prevents other tasks from attempting to control the axis. To attach an axis, issue the ATTACH command from the controlling task:

```
Program
       Attach

End Program
```

If no other task is attached to the axis, the axis is immediately attached. If another task has already attached the axis, an error is generated. A "Try…Catch" command is used to wait for an axis to be unattached. As long as the task has the axis attached, no other task can control the axis. When a task is finished with an axis, it should detach the axis with the DETACH command.

If you issue the DETACH command while profile motion for the axis is still in progress, the DETACH command delays execution until that motion is complete. If a task ends with an axis still attached, the axis is automatically detached.

One exception for the requirement that an axis be attached is if the motion command is issued from the terminal window. In this case, assuming the axis is not attached by any other task, the axis is automatically attached to the terminal window for the duration of the motion command.

Setting sys.conmode

To enable the SERVO**STAR** SC, SYS.CONMODE must equal 2. The following command sets the SERVO**STAR** SC in CONMODE 2 in the terminal window:

```
sys.conmode = 2
```

From a task or program, the following sets of commands ensure the SERVO**STAR** SC is in CONMODE 2. The IF statement must be used because if the drive is enabled, SYS.CONMODE can not be changed and a program error occurs.

```
If en <> 1 Then
       sys.conmode = 2
EndIf
```

Hardware Enable

To be in the enabled state there must be no errors and the hardware enable input (Connector C3 Pin 8) and the motion input (Connector C9 Pin 2) must be turned on.

Software Enable

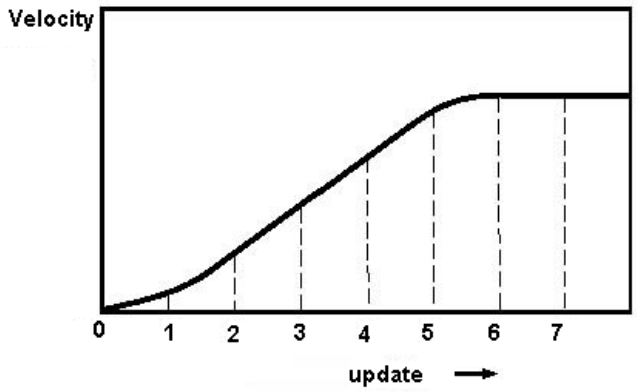
Now, enable the SERVO**STAR** SC by typing, EN = 1, in the terminal window.

Motion Flag

When an error occurs, the Motion Flag is often turned off. To turn the Motion Flag back on, you can either type in SYS.MOTION = 1 from the terminal (you can not do this in a task) or toggle the motion input (DIN1) at connector C9 pin 2.

Profile

The purpose of the motion generator is to convert motion commands (JOG or MOVE) into a regularly-updated series of position and velocity commands. That series is called a "motion profile." The position and velocity commands are sent to the controlled SERVO**STAR** SC and all servo loops (position, velocity, and current) are closed. The next figure shows the profile for the acceleration portion of a JOG command.



Update Rate

The profile must be updated at regular intervals. The SERVO**STAR** SC updates the profile at 1ms intervals.

PositionToGo

The axis property POSITIONTOGO (PTOGO) is the distance between the current POSITIONCOMMAND and the end of the profile.

Motion Buffering

The motion generator for the axis processees only one motion command at a time. This is because the axis processes only one position or velocity command at a time. However, you can send a second motion command to the generator where it is held pending until the current motion command is completed. This is called "buffering."

Motion buffering is controlled with the axis property, STARTTYPE. STARTTYPE can be set to GENERATORCOMPLETED so the buffered profile starts immediately after the current profile is completed. STARTTYPE can also be set to INPOSITION, where the buffered profile starts after the current profile is complete and the position feedback has settled out to the position command. In both cases, the second command is held in the motion generator to begin immediately after the appropriate conditions are met.

Alternatively, you can specify that the motion generator not buffer but process the new motion command immediately. This is useful when you want to make realtime changes to the profile (changing the end position of the current command). For this, you would set STARTTYPE to IMMEDIATE (or IMMED.) or SUPERIMMEDIATE (or SIMM).

Override vs. Permanent

An axis has numerous properties, such as acceleration and deceleration. These properties are normally set directly. For example:

```
Acc = 100
```

This setting is permanent and will persist until the next time the property is assigned a value. For convenience, the SERVO**STAR** SC also supports "override" values where the property is set as part of a command. In this case, the setting is used only for the current command. For example:

```
Acc = 100
Jog 1000 Acc = 10
```

Even though the acceleration rate of the axis is specified at 100 in the first line, the Jog accelerates the override value at 10. Motion commands that do not specify an override acceleration rate accelerate at the permanent value.

Override values are used extensively in motion commands. The values that can be overridden are specified in the description of the motion commands.

Acceleration Profile

The motion commands, MOVE and JOG, produce acceleration curves that are limited by the following axis properties:

ACCELERATION (ACC)

DECELERATION (DEC)

SMOOTHFACTOR (SMOOTH)

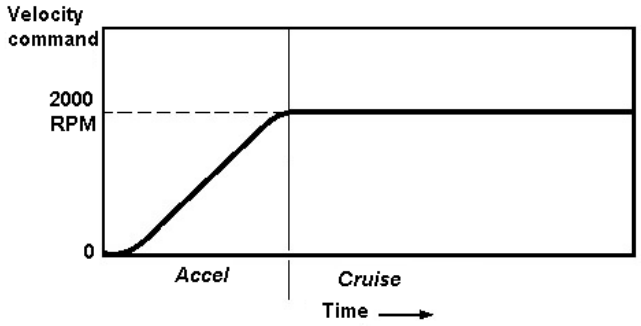
ACCELERATION and DECELERATION are limited to less than the acceleration capabilities of the motor and load. SMOOTH is used to help limit or reduce mechanical wear on the machine.

Jogging

JOG implicitly switches the system into digital velocity mode. When you want to command the motor to move at a constant velocity independent of the current position, use the JOG command. Velocity can be negative to produce motion in the reverse direction. For example,

```
Jog 2000
```

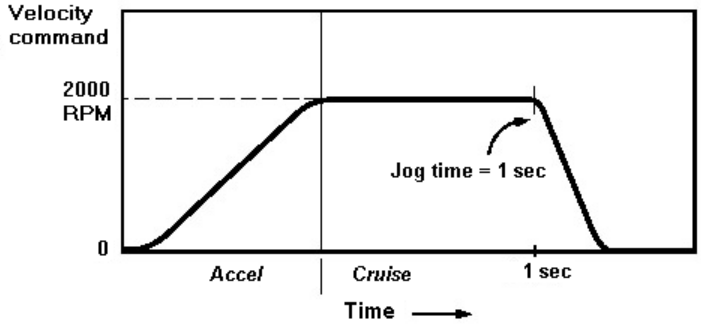
Produces the JOG profile below.



You can limit the amount of time the JOG command runs using the axis nodal-only property, TIMEJOG. TIMEJOG is specified in milliseconds. For example:

```
Jog 2000 TimeJog = 1000
```

Produces the JOG profile shown in the next figure.



When TIMEJOG = 1, the JOG command continues indefinitely. TIMEJOG defaults to –1.

Overriding Axis Properties In JOG

The following axis properties can be overridden as part of a JOG command:

- TIMEJOG
- ACCELERATION
- DECELERATION
- SMOOTHFACTOR
- JERK
- STARTTYPE

Stop

The STOP command is used to stop motion in the motion buffer.

Normally, STOP is set to stop motion immediately at the rate of the axis property, DECSTOP. However, you can modify the effects of STOP with the axis property, STOPTYPE. STOPTYPE can take two values:

STOPTYPE = IMMEDIATE or IMMED
Stop axis immediately at DECELMAX

STOPTYPE = ENDMOTION
Stop axis at end of current motion command and clear buffered motion

STOPTYPE defaults to IMMED. If STOPTYPE = ENDMOTION, the current MOVE is completed before the STOP is executed. Of course, for the JOG command with TIMEJOG = -1 (continue indefinitely), STOPTYPE = ENDMOTION has no meaning because the JOG never ends. If TIMEJOG is a positive number, using the STOP command with STOPTYPE = ENDMOTION halts the motion generator at the end of the JOG so any buffered moves are not generated.

Overriding Axis Properties In STOP

The STOP command uses the modal maximum deceleration and maximum jerk. The STOPTYPE property can be overridden as part of a STOP command.

Proceed

When a task stops motion in an axis it has attached, restarting the motion is simple. The logic is contained inside the task. However, when motion is stopped from another task, the situation is more complex. For example, the system should prevent any task from restarting the motion except the task that stopped it. To control the restarting of motion after a STOP, the SERVO**STAR** SC provides the PROCEED mechanism.

The PROCEED mechanism has two purposes: to enhance safety and to allow motion to continue along the original path. The safety enhancement is provided by not allowing motion on an axis to restart without the task that stopped the motion issuing a PROCEED command. The path control is provided by the axis PROCEEDTYPE property. You can set PROCEEDTYPE to continue to four modes. The key rules of operation are:

- When a task stops the motion of an axis to which it is attached, a PROCEED command is allowed, but not required.
- When a STOP command is issued from the terminal window, and the axis being stopped is attached to a task, motion is stopped and the task is suspended upon execution of the next motion or DETACH command. The task and the motion it commands can restart only after a PROCEED command is issued from the terminal window. Motion can be commanded from the terminal window before the PROCEED command is issued.

- When a STOP command is issued from one task, and the axis being stopped is attached to another task, motion is stopped and the task is suspended upon execution of the next motion or DETACH command. The task and the motion it commands can restart only after a PROCEED command is issued from the task that stopped motion. Motion cannot be commanded from the task that stopped motion.
- When a STOP command is issued from the terminal window for a MOVE command that was given from the terminal window, the motion is stopped and cannot proceed. In this case, the only available PROCEEDTYPE is CLEARMOTION.

There are three ways for the PROCEED command to restart motion. Use PROCEEDTYPE to specify how the motion generator should proceed:

PROCEEDTYPE = CONTINUE
This causes the motion generator to continue the motion command that was stopped.

PROCEEDTYPE = NEXTMOTION
This causes the motion generator to abort the current move and go directly to the move that is in the motion buffer.

PROCEEDTYPE = CLEARMOTION
This clears the motion buffer. All motion commands in the motion buffer are aborted. This is the default PROCEEDTYPE.

Move

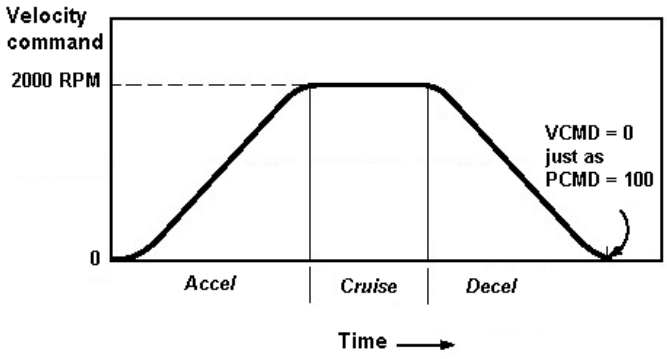
The MOVE command implicitly enters the system into digital position mode and is the most common of point-to-point moves. The basic MOVE is a three-segment motion:

ACCELERATE	Go from initial speed to VCRUISE
CRUISE	Continue at VCRUISE.
DECELERATE	Go from VCRUISE to VFINAL

The critical part of a three-segment MOVE is starting the deceleration at the right time so that the motor speed becomes zero (if VELOCITYFINAL is specified to be zero) just as the position command reaches the final position. For example:

```
Move 100  VCruise = 2000
```

Produces the following typical three-segment move profile.



Typical Three-Segment Move

Point-to-Point (PFINAL) Moves

POSITIONFINAL or PFINAL (the end of a move), is always specified in the MOVE command. The meaning of PFINAL depends on the axis property, ABSOLUTE or ABS. This allows point-to-point moves to be specified two ways:

Absolute Moves (ABSOLUTE = TRUE)
Final position is specified as actual motor position at the end of the move. The final position equals PFINAL.

Incremental (ABSOLUTE = FALSE)
Final position is referenced to the start position. The final position equals the sum of PFINAL and PCMD from the start of the move.

For example:

```
Absolute = TRUE
Move 100
```

Moves the axis to position 100. On the other hand:

```
Absolute = FALSE
Move 100
```

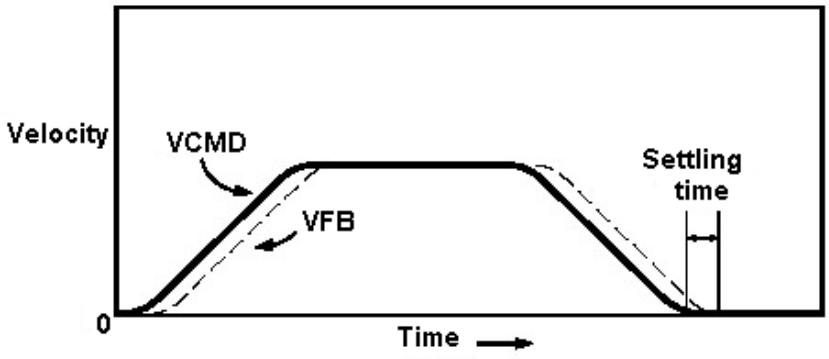
Moves the axis a distance of 100 units from the current position.

ABSOLUTE defaults to FALSE. You can change Absolute at any time, although the effect does not take place until you issue the next MOVE command.

Settling Time

The SERVO**STAR** SC actively watches to determine if the axis is settled into position. In almost all applications, the motor position feedback is slightly delayed from the position command. After a move is complete, some time is required for the actual position to settle out to the commanded position. This time is called "settling time."

For example, consider the point-to-point move illustrated below. The VELOCITYCOMMAND (VCMD) is shown in solid and VELOCITYFEEDBACK (VFB) is shown in dashed lines. The area between the two curves is the following error. As you can see, it takes a small amount of time at the end of the move for VFB to settle out to zero. The actual amount of time required for this varies from one system to another. Higher bandwidth systems have shorter settling times, but all systems need some time to settle. Typical times range from a few milliseconds to tens of milliseconds.



Point-to-Point Move with Following Error

Ideally, settling time allows the motor to move so as to approach zero position error. However, you must allow for the condition where the position error never quite reaches zero. On the SERVO**STAR** SC, you specify how low you consider to be low enough with the axis property POSITIONERRORSETTLE or PESETTLE. After the motion generator has completed a move ending with zero speed, it actively monitors the position error on the axis to see when it is between ±PESETTLE.

In some applications you must ensure that the position error remains below PESETTLE for a specified period of time before the axis is considered "settled." The SERVO**STAR** SC allows you to specify this time period with the axis property, TIMESETTLE or TSETTLE (in milliseconds). TIMESETTLEMAX sets the maximum time allowed for the axis to settle. If the position error exceeds PESETTLE during this time, the timer is reset. The flag, ISSETTLED, is TRUE (1) if position error is within PESETTLE range for the time specified by TSETTLE. Otherwise, it is FALSE (0).

Starting Point-to-Point Moves

Point-to-point moves begin immediately when the motion buffer is empty. If you need to delay the start of the next motion command, you have two options: use the DELAY command to insert a fixed delay time, or use STARTTYPE to delay, depending on a condition.

DELAY

Use the DELAY command to force the motion generator to wait a fixed period of time between moves. For example:

```
Move 100
Delay 1000
Move 200
```

Force a 1 second delay between the two moves. DELAY has units of milliseconds and must be greater than zero.

DELAY does not delay the execution of your program. If you want to delay your program, use the SLEEP command.

STARTTYPE

If you want to delay the start of a new move until a condition has been met by the previous move, use the axis property, STARTTYPE. There are four choices:

STARTTYPE = GENERATORCOMPLETED (GCOM)
When STARTTYPE = GCOM, the new Move starts as soon as the motion generator has completed the motion for the current move. GENERATORCOMPLETED and GCOM are constants equal to 3. This command incurs a system delay of 2 milliseconds.

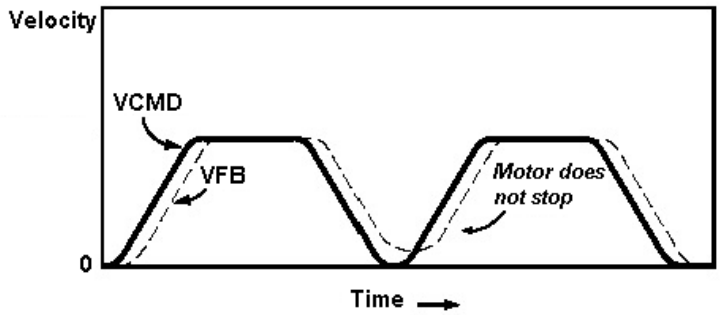
STARTTYPE = SETTLED
When STARTTYPE = SETTLED, the motion generator delays executing the new motion command until the current move has completed and the position error is settled to near zero. SETTLED is a constant equal to 2. This command is not subject to additional system delay, but any delay associated with the command is due to user-specified parameters for settling time.

STARTTYPE = IMMEDIATE (IMMED)
When STARTTYPE = IMMED, the new move overwrites the current move. This is used when making realtime changes to the profile, such as changing the end-point of the current move without bringing the system to rest. For example, registration applications frequently use this function. IMMEDIATE (or IMMED) is a constant equal to 1. This command incurs a system delay of 5 milliseconds per execution. This time is required to blend the previous move with the new move. If the next motion command is buffered, no additional delay is incurred.

STARTTYPE = SUPERIMMEDIATE (SIMM)
The SUPERIMMEDIATE type is a variation on the Immediate type. The main difference is that SUPERIMMEDIATE eliminates the 5-millisecond delay by doing pre-calculation online, rather than offline in the Motion Manager. SUPERIMMEDIATE and SIMM are constants equal to 5.

CHAINING ZERO-TERMINATED MOVES

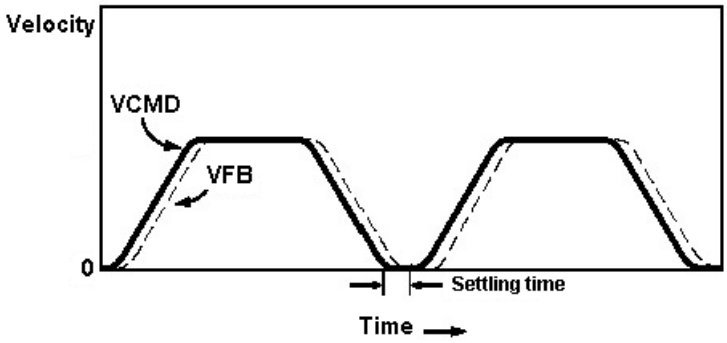
When chaining multiple moves that all end at zero speed, you normally want STARTTYPE = INPOS. This forces the motion generator to wait for the position to settle out before starting the next move. If the position profile starts too soon on the second move, the motor may never come to rest. The figure below shows this problem when STARTTYPE is incorrectly set to GCOM rather than INPOS.



Multiple Point-to-Point Moves Started Too Quickly

As you can see, the motor speed never gets to zero because the second move takes the velocity command positive before the velocity feedback can settle to zero.

Normally, the velocity should reach zero before the second move starts. To do this, the controller needs to wait for the axis position error to be settled before starting the second move. For this, set STARTTYPE to INPOS. This is shown in the next figure.



Multiple Point-to-Point Moves Started Correctly

As you can see, the motor does come to rest because the SERVO**STAR** SC waits for the following error to be small enough before proceeding to the next command.

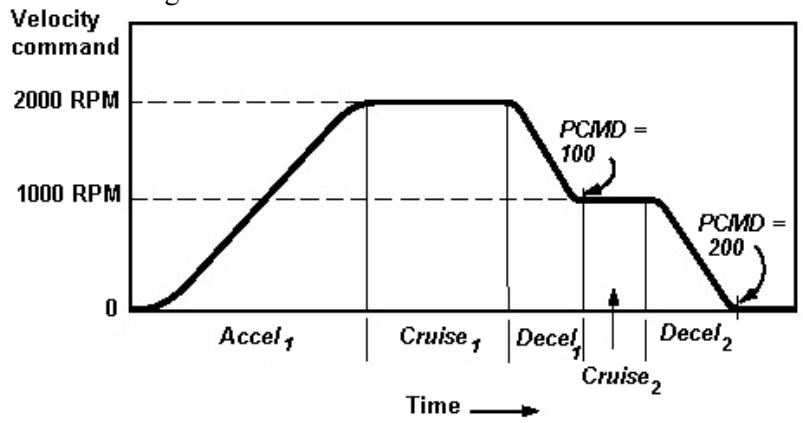
MULTI-STEP MOVES

Combining non-zero end-point moves in the motion buffer produces multi-step moves. This is accomplished by adding the property, VELOCITYFINAL (or VFINAL) to the MOVE command. The first move in a series of MOVE commands must be set to VFINAL=0. If not, the last part of the motion profile abruptly decelerates to zero and can halt task execution.

For example:

```
Program

  Attach
  Call AxisSetup

  Sys.Conmode = 2
  En = On

  Abs = Off
  StartType = GCom

  'Do Two-step Move
  Move 100 VCruise = 20 VFinal = 10
  Move 100 VCruise = 10 VFinal =  0

  'Wait for move to complete and axis to settle out
  While IsSettled = 0
    Sleep 10  'Sleep in loop to keep from overloading CPU resources
  End While

  'Disable, detach and exit
  En = Off
  Detach

End Program
```

Produces the profile shown in the next figure.



Two-step Move

Notice in the above example that STARTTYPE was set to GCOM. This means that the second move should begin when the first move is generated. You must use STARTTYPE = GCOM when chaining a non-zero end-point move to another move. You cannot use STARTTYPE = IMMED or SIMM because the new move overwrites the old move. You cannot use STARTTYPE = INPOS because you want the second part of the profile to begin immediately after the first. The delay from waiting for "In Position" adds error if the second move is incremental. In fact, for any move type, the move may be delayed indefinitely because moving motors never come "in position" unless the system is specifically tuned to do that.

You can combine non-zero end-point moves to produce profiles with as many steps as you need. However, you need to be aware of a few restrictions:

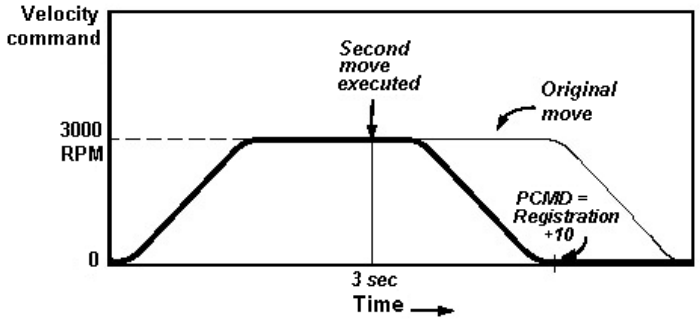
- STARTTYPE for all non-zero end-point moves (that is, VELOCITYFINAL < > 0) must be GCOM.
- VELOCITYCRUISE and VELOCITYFINAL are always positive. The direction is set by target position.
- PFINAL of the succeeding move must be far enough in front of the PFINAL of the current move so that the profile is possible with the acceleration limits. The generated profile always reaches the target position. If the acceleration and smooth limitations prevent attaining the required final velocity, the motion is terminated with the final velocity as close as possible to the required value.

REALTIME CHANGES

You can change the end position or the end velocity of a move that is in progress. You do this by setting STARTTYPE = IMMED and issuing the second move. For example, if you want to issue a MOVE, wait a few seconds then change the end position without changing the velocity. You would write:

```
StartType = Immed
Absolute = On
Move 150
Sleep 3000
Move 100
```

This generates the profile shown in the next figure.



As a second example, if you wanted to change the cruise velocity without changing the end-position, you would write:

```
StartType = IMMED
Move 150
Sleep 3000
VCruise = 1500
Move 150
```

You can change the final position, cruise velocity, and final velocity of any move that is executing. However, you must observe a few rules:

- STARTTYPE must be IMMED or SIMM.
- Direction is set according to the target position or the sign of the velocity in the case of JOG motion.
- New and old VELOCITYCRUISE and VELOCITYFINAL must all have the same sign.
- If the succeeding MOVE changes PFINAL or VFINAL, it must remain possible to create the profile with the axis acceleration limits.

Overriding Axis Properties in MOVE

You can override the following axis properties as part of a MOVE command:

ABSOLUTE
ACCELERATION
DECELERATION
SMOOTHFACTOR
VELOCITYCRUISE
STARTTYPE

```
VCruise = 100              'Sets MOVE velocity to 100
Move 1500                  'Move at VCruise=100
Sleep 3000                 'Delay task execution
Move 1500 VCruise = 150    'Moves at VCruise = 150
Sleep 3000                 'Delay task execution
Move 1500                  'Move at VCruise=100
```

Velocity Override

The SERVO**STAR** SC provides the ability to change the speed of all the motion commands of the SERVO**STAR** SC. This capability is used extensively in machine development as it enables you to adjust the entire machine speed in a single command. Since the command can be issued from the terminal, you can observe operation of the machine at a variety of speeds without modifying your program.

The axis on a SERVO**STAR** SC is controlled with the system property VELOCITYOVERRIDE, or VORD. For example:

```
VelocityOverride = 25          'Short form is VORD=25
```

Immediately reduces the velocity commands of all currently executing MOVE and JOG commands (as well as any subsequent commands) to 25 percent of the current values. Since VORD controls the velocities, the acceleration rates of the axis are proportionally adjusted. The final positions of MOVE commands are not effected.

Motion

The SERVO**STAR** SC provides several ways to initiate motion. The most common are discussed below.

Position Capture

The SERVO**STAR** SC implements the position capture functionality. This is commonly used in homing and registration procedures.

The SERVO**STAR** SC provides various commands to control the capture procedure. The SERVO**STAR** SC provides three digital inputs (IN1, IN2, and IN3) on connector C3, any one of which can be designated for the capture trigger signal. Setting the property IN1MODE, IN2MODE, or IN3MODE equal to 16 defines IN1,IN2, or IN3 on connector C3 as the capture hardware input.

To execute a capture procedure based on a input the following must be performed:

- Define IN1,IN2, and IN3 as the capture input (INxMODE)
- Set which edge to capture position (CAPTUREPOLARITY)
- Start the CAPTURE procedure (CAPTURE)
- Arm the position capture mechanism (CAPTUREARM)
- Wait for capture to occur (CAPTURESTATUS)
- Perform follow up motion or action (CAPTUREDPOSTION)
- Reset the capture position mechanism (CAPTURREARM)

Homing

Most machines need to come to a reference position after they power up. This is usually straightforward, as the reference position is usually a fixed offset from the position feedback device. However, for geared or belted machines, or machines driven by a ball screw, the procedure is more complicated as the relationship of the motor and load is often unknown at power up. In these cases, it is common to place a mechanical limit or proximity switch on the machine to indicate the reference position.

The SERVO**STAR** SC supports various types of homing(HOMETYPE) as well as offset moves (HOMEOFFSET) from the machines home switch and/or marker pulse. Additionally the value for the feedback position PFB can be set (HOMEDISTANCE) after homing and any offset move is made.

You start the homing procedure on the axis by issuing the command, HOME. The SERVO**STAR** SC immediately begins to execute the homing procedure. Refer to Appendix C for a sample HOME program that is set up as a subroutine.

Registration

Registration applications are those that start motion and then modify the profile, based on a subsequent event, typically a change in state of a digital input.. These applications generally involve discrete product processing common in the packaging, printing and converting industries. Usually, sensing a position on the product being processed generates the event. Typically, a mark is printed on the product and detected by an optical sensor. The mark is normally referred to as a "registration mark."

A common registration application is cutting a product package such as a bag or label from a web (reel). The web is printed with many copies of the product package. The controller begins to unwind the product package from the web. During this motion, the controller waits for a registration mark to be detected. After the mark is detected, the profile is modified, based on the position at which the mark was detected. Commonly, the profile comes to rest a fixed distance after the mark position where the package can be cut. There are numerous variations of this type of application. The key common element is that the end position of the move cannot be calculated until after the move starts. This requires that the profile be started and then modified on-the-fly.

The SERVO**STAR** SC supports these applications by combining the ability to change position end-points on-the-fly and the ability to command the SERVO**STAR** SC to capture a position. Registration is similar to homing except that in homing, you go back to the mark, but in registration you go forward to a fixed distance after the mark. Refer to Appendix C for a sample registration program.

Master/Slave

Master/Slave **MOTIONLINK**s the position of one axis (the slave) to the position of another (the master). The master axis is the axis controlled by the SERVO**STAR** SC. External axes are connected to the SERVO**STAR** SC via the external encoder input.

There are two types of master/slave motion: gearing and CAMming. Gearing is used when the position of the slave should be proportional to the position of the master. CAMming is used when the relationship is more complex. CAMming allows you to specify a one-to-one mapping of the positions of the master and slave. The master signal for master/slave motion is an encoder signal connected to the external encoder input (connector C5) of a SERVO**STAR** SC.

Velocity Override

If the master VELOCITYOVERRIDE is changed, the slave velocity is changed accordingly. In the case of a relative move, the slave moves according to its new VELOCITYOVERRIDE. VELOCITYOVERRIDE has no effect on the master signal for a slaved axis. It also has no effect on the gear ratio or CAM table. VELOCITYOVERRIDE modifies the velocity of incremental move, which can be added onto geared or CAMmed motion.

Gearing

When two axes are "geared", the position command of the slave is proportional to the master signal. This is shown in the next figure.

Gearing works in both directions of master travel. The term "proportional" is applied loosely. Actually, the offset between the master and slave axes is saved at the time gearing is enabled and is maintained throughout gearing operations.

ENABLING AND DISABLING GEARING

To enable gearing, set the MASTERSOURCE to PEXT and the MASTERSLAVE to the constant GEAR:

```
MasterSource = PEXT          'Sets the MasterSource to connector C5
Slave = GEAR                 'Enable gearing
```

Enabling gearing during an absolute or relative move generates an error. An error is also generated if an attempt is made to enable gearing during CAMming. Incremental moves are allowed when gearing is enabled as well as during gearing.

To disable gearing, set SLAVE to OFF:

```
Slave  = OFF                  'Disable gearing
```

When gearing is disabled, the velocity command of the slave axis is decelerated to zero at the rate of the axis property DECSTOP.

Disabling an axis does not automatically disable gearing. You do not need to enable gearing again when the axis is re-enabled. However, issuing a JOG or STOP command for the axis disables gearing.

Issuing a stop command turns gearing off immediately. The velocity command of the axis is decelerated to zero at the rate set by the axis propert,y DECSTOP.

For a JOG command, gearing is disabled immediately and the movement is controlled by the JOG profile.

GEARRATIO

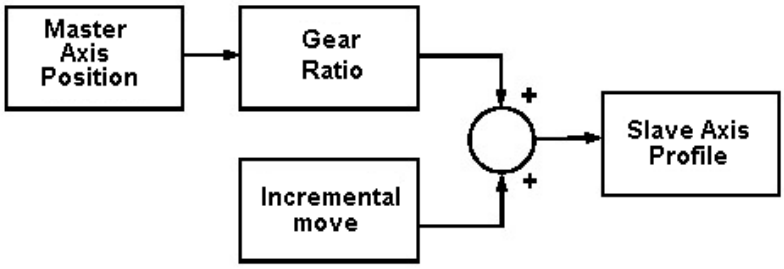
Set the proportional constant between the master and slave with the axis property, GEARRATIO. For example:

```
GearRatio = 0.5              'Slave goes half the speed of master
```

GEARRATIO is a double-precision floating-point number. It can be set to less than zero to reverse the direction of the slave.

INCREMENTAL MOVES WITH GEARING

The SERVO**STAR** SC supports incremental moves with gearing. In this case, the profile of the slave axis is the sum of two profiles: the gearing profile and the profile of the incremental move, as shown in the next figure.



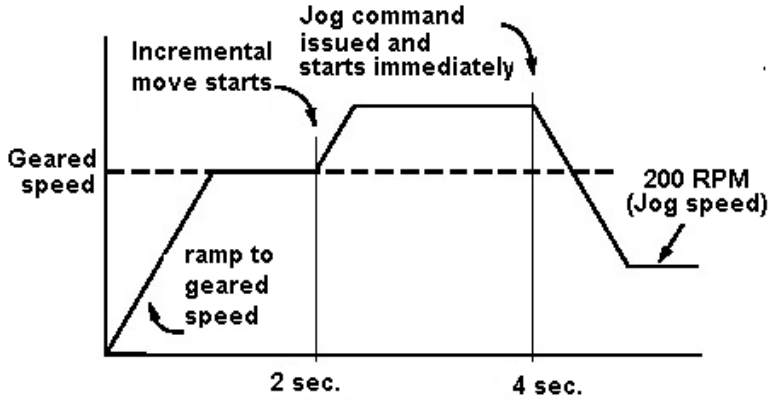
You can use incremental moves with gearing, much as you use them without gearing. Issuing an absolute MOVE with gearing enabled generates an error.

If you issue a STOP command to a geared axis, which is executing incremental moves, gearing is turned off immediately and the velocity commanded from gearing ramps to zero at DECSTOP. On the other hand, the termination of the incremental MOVE profile and subsequent launch of the JOG is subject to STOPTYPE.

Issuing a JOG (when gearing with an incremental MOVE), is similar to issuing a Jog with just gearing. The main difference is that the launch of the JOG profile is subject to STARTTYPE. If STARTTYPE is IMMEDIATE (IMMED) or SUPERIMMEDIATE (SIMM), the JOG begins immediately. The speed from the incremental MOVE ramps up or down to the specified JOG speed according to DEC or ACC. If STARTTYPE is GCOM, SYNC or INPOS, the JOG profile is delayed. The following example issues a Jog with an immediate start (STARTTYPE = IMMEDIATE).

```
Jog 100 TimeJog=5000
Sleep 2000
StartType=IMMED
Jog 200
```

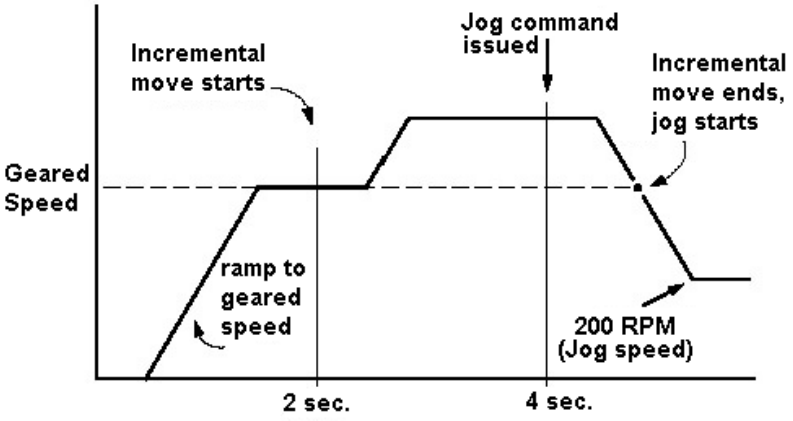
The graph in below shows the resulting profile.



The next example is similar to the above example, but with the starting after the incremental MOVE is complete (STARTTYPE = GCOM):

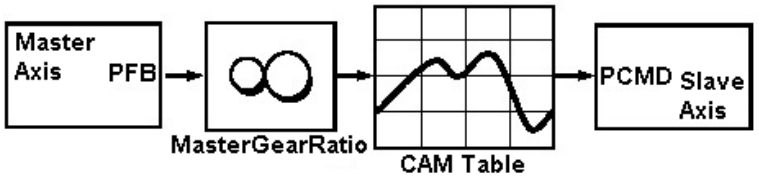
```
Jog 100 TimeJog=5000
Sleep 2000
StartType=GCOM
Jog 200
```

The following graph shows the resulting profile.



CAMming

CAMming is an extension of gearing. It can be thought of as gearing where the gear ratio varies according to the master position. The master position, optionally run through a gear ratio, drives the input to a CAM table. The CAM table is a list of point pairs, which map the master position to the slave position command. The output of the CAM table provides the slave axis position command. The next figure shows the operation of CAMming.



KEY FEATURES OF CAMMING

The SERVO**STAR** SC provides CAMming with the following features:

- CAMming supports motion in both directions. The master can run indefinitely in either positive or negative directions without accumulation of error.
- CAM tables can be driven by an encoder brought in through the SERVO**STAR** SC external encoder input.
- CAM tables are two-dimensional. The points may be irregularly spaced. Linear interpolation is used between points.
- CAM profiles need not return to the original position at the end of the profile.
- CAM tables can be calculated off-line and stored or they can be calculated "on the fly".
- CAMs can cycle indefinitely or they can be specified to cycle a specified number of times.
- Multiple CAMs may be defined and linked together automatically.
- The master position is processed through the Master-Slave gear ratio (GEARRATIO).
- You can issue incremental Moves, which are summed with the CAM profile to form the slave position command.

CAM AND GEARRATIO

As with gearing, in CAMming you set the proportional constant between the master and slave with the axis property GEARRATIO. If GEARRATIO is 1, the table is matched one-to-one to the master axis. If it is less than one, the table is driven at a speed slower than the master is moving. For example, to run the table at half the speed of the master:

```
GearRatio = 0.5          'Drive CAM table at half speed
```

In this case, the master would turn two units to drive the table one unit. GEARRATIO is a double-precision floating-point number. To reverse the direction of the slave, enter the ratio as a negative number.

INCREMENTAL MOVES WITH CAMMING

You can issue incremental moves to the slave with CAMming in the same way as with gearing. In this case, the profile from the incremental move is summed with the profile for gearing to form the slave position command.

CAM TABLES

A CAM table is a list of master/slave point pairs that define a one-to-one mapping of the master axis to the slave axis. The user specifies the number of point pairs. It may be as large or as small as the application requires. The following is a typical CAM table:

```
CAM1.MasterData[1]=10
CAM1.SlaveData[1]=20
CAM1.MasterData[2]=20
CAM1.SlaveData[2]=40
CAM1.MasterData[3]=30
CAM1.SlaveData[3]=60
CAM1.MasterData[4]=40
CAM1.SlaveData[4]=80
CAM1.MasterData[5]=50
CAM1.SlaveData[5]=100
```

The individual points in the table may be given at irregular intervals. This is useful if the CAM profile has long smooth sections and sharp turns. Irregular intervals allow you to use a few points for the smooth section and many for the sharp turns. Although the interval of master position may be irregular, it must always be greater than zero. That is, the master position must be "monotonically increasing" or “monotonically decreasing” in the CAM table. The slave position is not limited by this requirement.

INTERPOLATION

In the unusual case that the master position is located exactly on one of the master position points, the slave position command is the corresponding point of the master/slave point pair. When the master is located between two points, the SERVO**STAR** SC uses linear interpolation to calculate the position.

INCREMENTAL OR ABSOLUTE

CAM tables are incremental in one aspect and absolute in another. Within the CAM table, the positions are absolute. Suppose we wanted a CAM table with 5 points, with each point of the master separated by 10 units, and slave positions separated by 20 units. The table (with the assumed name CAM1) would be:

```
CAM1.MasterData[1]=10
CAM1.SlaveData[1]=20
CAM1.MasterData[2]=20
CAM1.SlaveData[2]=40
CAM1.MasterData[3]=30
CAM1.SlaveData[3]=60
CAM1.MasterData[4]=40
CAM1.SlaveData[4]=80
CAM1.MasterData[5]=50
CAM1.SlaveData[5]=100
```

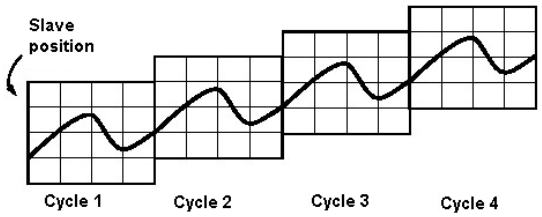
As you can see, the master and slave positions are absolute within the table. This method is not a valid way of entering CAM table items, but it is an illustration of the table entries.

CAM tables are incremental in regard to the start point. The start position in a CAM table is always offset to the master and slave positions when the CAM table is enabled. The operation of the table above is identical to the table below where the point pairs are adjusted to allow the first point to change from (10, 20) to (0, 0):

CAM1.MasterData[1]=10
CAM1.SlaveData[1]=20
CAM1.MasterData[2]=20
CAM1.SlaveData[2]=40
CAM1.MasterData[3]=30
CAM1.SlaveData[3]=60
CAM1.MasterData[4]=40
CAM1.SlaveData[4]=80
CAM1.MasterData[5]=50
CAM1.SlaveData[5]=100

NET MOTION IN A CAM TABLE

CAM tables can be specified to command "net motion." That is, the start point and end point of the CAM table can be different. The reason CAM tables are incremental or decremental with regard to starting points is to allow a CAM table to generate net motion while still allowing it to be called many times in succession. Consider the following graph that shows just such a CAM table with net motion executed four cycles:



As you can see, each successive cycle ratchets the slave position up. Thus the starting position in the table cannot be equal to the actual slave position. Otherwise, the CAM slave position (fixed in the CAM table) and the actual slave position (ratcheting up each cycle) could not be matched for more than one cycle. Almost always the actual slave position and the current CAM slave value are not equal.

CAM TABLE LENGTH

The distance to be moved is specified indirectly through the CAM table data. The total displacement of one cycle of the master is the master position of the last point pair less the master position of the first, divided by the axis property GearRatio. In many applications, you must take care to ensure that this quantity is precise or the master may appear to creep over time. The SERVO**STAR** SC uses double precision math (16 digits of accuracy) for CAM calculations. You should use the SERVO**STAR** SC to calculate mathematical expressions to obtain the greatest possible precision. For example, if GEARRATIO is 1:3, use the SERVO**STAR** SC to calculate the value rather than rounding a previously calculated value:

```
MasterGearRatio = 1/3            'Good: 1/3 accurate to 16 places
MasterGearRatio = 0.33333        'Poor: 1/3 accurate to 5 places
```

The "net motion" commanded to the slave is the slave position of the last point pair less that of the first. If you do not want net motion in the slave then you must assign the slave positions of the first and last point pairs to be identical.

CALCULATING OFFSETS IN CAM PROFILES

At the point when CAMming is enabled, the SERVO**STAR** SC takes a snapshot of the master position feedback (or position command or position external, depending on master definition) and slave position command and offsets all entries in the CAM table for the master and slave. CAM tables are incremental with respect to the actual position of the axes at the time CAMming is started. For example, suppose a CAM table starts at (0, 0) but when the CAMming is enabled, and the master position is 1000, and the slave commanded position (PCMD) was -2000. In this case, each point pair in the table would be implicitly adjusted by (1000, -2000) throughout the CAM cycle.

After each CAM cycle, the offset is calculated to the master and slave. The CAM master position is adjusted for the difference between the master position at the starting and ending points. If the CAM table has "net motion", each subsequent cycle adds the net motion to the slave commanded position.

ROTARY MASTER AND/OR SLAVE

Both the master and slave can be in rotary mode. This does not affect CAMming operation.

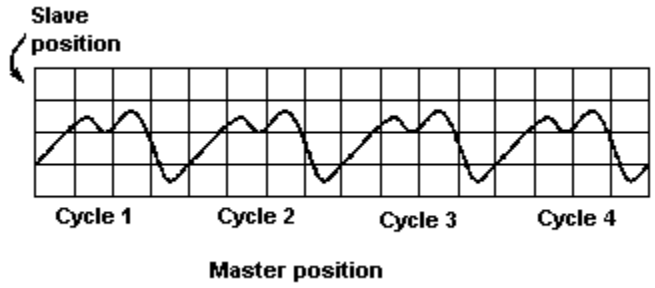
RUNNING CAM CYCLES

One CAM cycle is the operation of moving the master position feedback through an entire CAM table. Alternatively, instead of using master position feedback, you can also use position command or position external. At the end of a CAM cycle, the CAM can end or it can transition to the next cycle. The next cycle can be simply to repeat the current CAM or it can be another CAM. The number of times a CAM is repeated is controlled by the CAM property, Cycles. Other CAMs can be linked in automatically with the Next and Previous properties.

When the master is moving forward, the CAM position feedback counts up; the cycle ends when the master position is greater than the last entry in the CAM table. At this point, the CAM moves to the next cycle. Similarly, when the master is counting down, the cycle ends when master position is less then the first entry in the table; here, the CAM moves to the previous cycle. It all depends on whether the table is increasing or decreasing. If there is no next (or previous) CAM cycle, CAMming is disabled at this point.

REPEATING THE TABLE

Whether or not the CAM cycle repeats, and the number of times it should repeat, is specified in the CAM property Cycles. You can enter "1" to indicate that the CAM should not repeat (that is, run only once), -1 to indicate the CAM should run indefinitely, or a positive number that indicates the number of times the cycle should run. For example, suppose you want a CAM to run four cycles as shown in the next graph.



In this case, you would use the following line:

```
CAM1.CYCLE = 4                ' Run CAM1 four cycles
```

If you want a CAM to run the same table an indefinite number of times, set CYCLES to -1:

```
CAM1.CYCLE = -1               'Run the CAM indefinitely.
```

CASCYCLE

You can monitor the number of cycles that have been run with the axis property, CASCYCLE. When the master is moving forward, CASCYCLE counts up. When CASCYCLE reaches CYCLE, the axis transitions to the next CAM if there is one. Similarly, when the master is running backward, when CASCYCLE reaches 0 the cycle transitions to the previous CAM, if there is one.

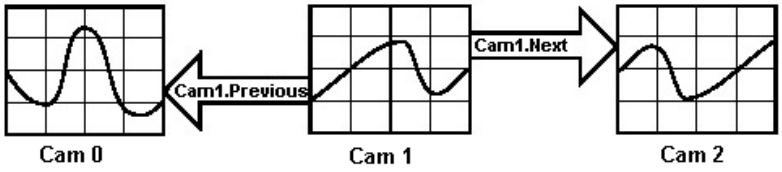
LINKING MULTIPLE CAM TABLES

The SERVO**STAR** SC allows you to specify multiple CAM tables at the same time. The maximum number of CAM tables is 256. The SERVO**STAR** SC allows you to link CAM tables together using two properties of the current CAM table: NEXT and PREVIOUS. These properties allow the automatic transition from one CAM table to another without the accumulation of error in the master or slave positions.

To link one table to another, assign the Next and Previous properties to the desired CAM table. For example:

```
StartCAM.Next = ForwardCAM
StartCAM.Previous = ReverseCAM
```

The linking of the three CAMs (REVERSECAM, STARTCAM, FORWARDCAM) is shown in the next figure.



The NEXT and PREVIOUS properties may be set independently. The NEXT and PREVIOUS pointers of a CAM table may point to itself. This has the effect of causing indefinite cycling of the CAM table in that direction. The NEXT and PREVIOUS pointers may not be changed dynamically; the effect will take place at the next transition of the CAM table. If you do not want to link to another CAM at the end of the current CAM's Cycle, then set NEXTor PREVIOUS to the constant, NULL.

LINKED CAMS AND CYCLE

If you have linked CAMs, the CAM cycle still runs the number of times specified in the CAM property Cycles. For example, assume a forward-rotating master is driving CAM1. If CAM1.Next is CAM2 and CAM1.Cycles = 4, CAM1 will cycle 4 times before transitioning to CAM2. Similarly, if the master is rotating backwards, and CAM1.PREVIOUS is CAM0, CAM1 will cycle 4 times before transitioning to CAM0.

EXHAUSTING CYCLES

If the CAM has run enough to exhaust the number of cycles, and no CAMs are linked to the current CAM, the axis automatically disables CAMming and decelerates at maximum deceleration to zero velocity.

CREATING CAM TABLES

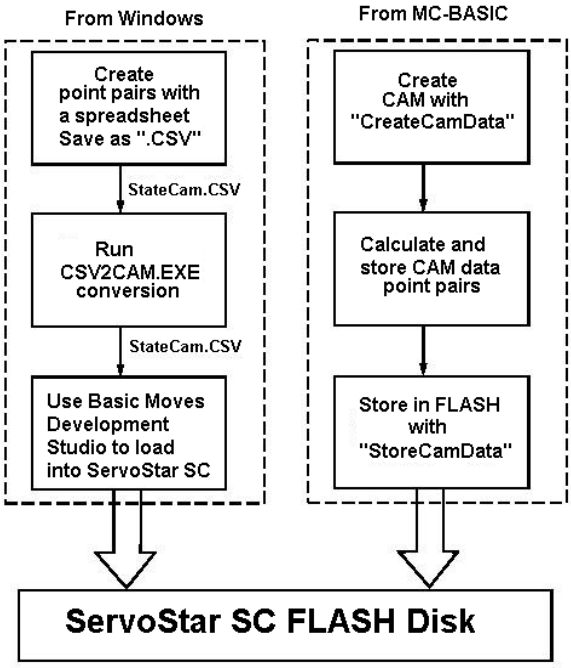
CAM tables contain the point pairs that map the master position feedback to the slave position command. CAM tables are only a part of CAMs. Other properties of CAMs include CYCLE, NEXT, and PREVIOUS. Only the table portion of CAMs are stored permanently in the SERVO**STAR** SC flash disk. The other properties must be set during program operation.

There are two types of CAM tables: static and dynamic. Static tables are built and stored in flash disk to be used later. Dynamic CAM tables are built at run time. You should choose a type based on your application.

STATIC CAM TABLES

Static CAM tables are built off-line and stored permanently in the SC flash disk. They have the advantage of allowing you to use advanced tools to graphically build and inspect the CAM table, and do not take away from system processing resources at run time. Static CAM tables are usually the best choice when the CAM table does not change during normal operation.

You can build static tables two ways. First you can use Windows tools to build the table, and then use SERVO**STAR** SC tools to convert and store the table. You can also build static CAM tables inside the SERVO**STAR** SC. Both methods are shown in the next figure.



Normally, if you are using a static table, it is recommended that you use the Windows-based tools. This method is simpler and you have more tools available with which to graph and analyze the CAM table.

BUILDING STATIC TABLES IN WINDOWS

To build a table in Windows, follow this procedure:

1. Globally allocate CAM storage.
2. Create point pairs with a Windows program such as Excel.
3. Save the point pairs as a "Comma Separated Variable" (CSV) file.
4. Insert the CSV file as part of the project.
5. BASIC Moves and MotionSuite automatically stores the table into SERVO**STAR** SC flash disk.

CREATE POINT PAIRS

You can create point pairs using windows applications such as a text editor, spread sheets (Excel), or mathematical programs (MatLab). For spreadsheets, create a new sheet with two columns and one row for each point pair; do not include a row for headings and do not include any other information within the sheet. The first element of each row should be the master position; the second should be the corresponding slave position. Recall that the master position must increase or decrease with each point pair as long as it is monotonic; its value cannot change directions (i. e., increase and then decrease) or even stay the same.

SAVE TO CSV

When the table is complete, save to a "Comma Separated Variable" (CSV) file. This is a text file format with one line for each point pair and commas between the master and slave values. For a five-point CAM table, the CSV file would look like:

10, 20
20, 40
30, 60
40, 80
50, 100

When you add the .CSV file to your project, BASIC Moves or MotionSuite converts it to a CAM table and stores it on the SERVO**STAR** SC.

BUILDING STATIC TABLES IN MC-BASIC

To build a table in the SERVO**STAR** SC using MC-BASIC, use the following procedure:

1. Globally allocate CAM storage
2. Create the CAM data table
3. Calculate the point pairs
4. Store the table in SC flash disk
5. Delete the CAM (optional)

GLOBALLY ALLOCATE CAM STORAGE

First globally allocate CAM storage. Use the COMMON command:

```
Common Shared CAM1 as CAM
```

(Here, CAM1 is assumed as the name. Replace CAM1 with any legal SERVO**STAR** SC name). The easiest way to do this is to include this line in Config.Prg and reboot the SERVO**STAR** SC.

The alternative is to enter this at the terminal window. Remember that the effect of this line persists only until the SERVO**STAR** SC resets.

CREATE THE CAM DATA TABLE

Now you must create storage for the table of point pairs. First, select a task that will be used to calculate the point pairs. Now, enter the following line in the that task:

```
Program CAMMaker
CreateCAMData 5 CAM1
```

This allocates 5 data points for CAM1. Essentially two arrays (of size 5) are created that are named CAM1.MASTERDATA and CAM1.SLAVEDATA.

CALCULATE THE POINTS PAIRS

Calculate the point pairs. You can build loops and iterate through the array. The array is 1 based and you will receive an error if you go outside the bounds of this array. For this example, the points are simply loaded:

CAM1.MasterData[1]=10
CAM1.SlaveData[1]=20
CAM1.MasterData[2]=20
CAM1.SlaveData[2]=40
CAM1.MasterData[3]=30
CAM1.SlaveData[3]=60
CAM1.MasterData[4]=40
CAM1.SlaveData[4]=80
CAM1.MasterData[5]=50
CAM1.SlaveData[5]=100

STORE THE TABLE IN FLASH

Finally, store the table on the flash disk. Use the following line:

```
StoreCAMData MyCAM.CAM CAM1
```

This stores the CAM as "MyCAM.CAM" in flash memory. Note that the name of the file follows the 8.3 format. This means that the filename cannot have more than 8 characters in the main part and 3 characters in the extension. It is recommended that you always use ".CAM" as the extension for CAM files. This file can be loaded at any time using LoadCAMData.

DELETING CAMS (OPTIONAL)

If you created the CAM storage (Common Shared...) from the terminal, you can delete it with:

DeleteCAM CAM1

A CAM table cannot be deleted if it is in use or if the task that it is attached to is loaded.

REVIEW ENTIRE PROCESS

The entire process is: from the terminal or Config.Prg:

```
Common Shared CAM1 as CAM
```

From the task that calculates the CAM:

```
Program CAMMaker
   CreateCAMData 5 CAM1
   CAM1.MasterData[1]=10
   CAM1.SlaveData[1]=20
   CAM1.MasterData[2]=20
   CAM1.SlaveData[2]=40
   CAM1.MasterData[3]=30
   CAM1.SlaveData[3]=60
   CAM1.MasterData[4]=40
   CAM1.SlaveData[4]=80
   CAM1.MasterData[5]=50
   CAM1.SlaveData[5]=100
   StoreCAMData MyCAM.CAM CAM1
Program End
```

DYNAMIC CAM TABLES

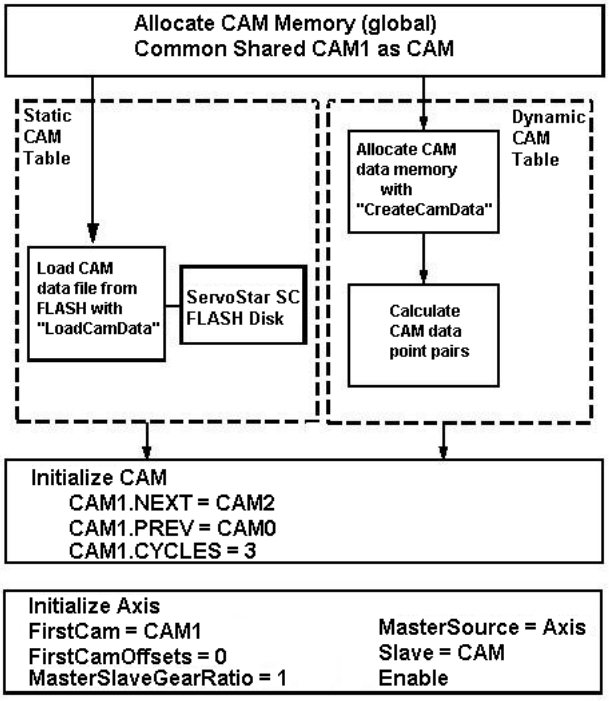
CAM tables can also be calculated dynamically. This allows you to build a CAM profile based on events that occur during operation such as the current position or speed of an axis. This process is identical to the process of creating a static CAM table from MC-BASIC except you do not use the STORECAMDATA command to place the data in flash. Instead, you use the CAM table directly.

OPERATING CAMS

The step-by-step process to use a CAM is:

- Allocate space for the CAM
- Load a static CAM data array or create a dynamic one
- Initialize other elements of the CAM
- Initialize the axis
- Enable the axis
- Read dynamic information from the CAM as needed

Throughout this section we will use CAM1 as our example. This process is shown the the following flowchart.



ALLOCATE SPACE

First, allocate space for the CAM using the COMMON command. All CAMs are global, so this line must be located in Config.Prg. For example, enter this line in Config.Prg:

```
Common Shared CAM1 as CAM
```

LOAD CAM DATA

Now, load the CAM data. This can be done in one of two ways. For a static CAM data array, which has been calculated and stored in the SERVO**STAR** SC, use the LOADCAMDATA command:

```
LoadCAMData MyCAM.CAM CAM1
```

LOADING A DYNAMIC CAM

You can dynamically calculate the CAM. This is a two step process. First you must create space for the data array and then you must fill that space. For example, for a five element, you could enter:

```
CreateCAMData 5 CAM1

CAM1.MasterData[1]=10
CAM1.SlaveData[1]=20
CAM1.MasterData[2]=20
CAM1.SlaveData[2]=40
CAM1.MasterData[3]=30
CAM1.SlaveData[3]=60
CAM1.MasterData[4]=40
CAM1.SlaveData[4]=80
CAM1.MasterData[5]=50
CAM1.SlaveData[5]=100
```

This is similar to building a static CAM table in the SERVO**STAR** SC. The difference is this CAM table is never stored to or loaded from flash.

INITIALIZE ELEMENTS OF THE CAM

Now, initialize other elements of the CAM. For example:

```
CAM1.Next = CAM2
CAM1.Previous = CAM0
CAM1.Cycles = 3
```

INITIALIZE THE AXIS

Initialize the elements of the axis that are related to CAMming. First set the property, FIRSTCAM, to point to the first CAM that the axis should follow:

```
FirstCAM = CAM1
```

Then, set the master position within the CAM at which you wish to start. For example, you may have a CAM table where the master position goes from 0 to 100.00. The correct starting position for the CAM might be 33.333. In this case, enter:

```
FirstCAMOffset = 33.333
```

FIRSTCAMOFFSET must be within the range of the master position. Remember that FIRSTCAMOFFSET defaults to 0. If the master range does not include zero, an error will be generated if you do not set FIRSTCAMOFFSET.

Set the GEARRATIO and name the source of the master signal:

```
GearRatio = 1.00
MasterSource = PEXT
```

Finally, set the SLAVE property to CAM:

```
Slave = CAM
```

The entire process is shown below for the case of a static CAM table:

In Config.Prg:

```
Common Shared CAM1 as CAM
```

In any other task:

```
Program

       LoadCAMData StaticCAM.CAM CAM1

       CAM1.Next = CAM2
       CAM1.Previous = CAM0
       CAM1.Cycles = 3

       FirstCAM = CAM1
       FirstCAMOffset = 33.33
       MasterGearRatio = 1.00
       MasterSource = PEXT
       Slave = CAM

End Program
```

ENABLE THE AXIS

At this point CAMming is enabled. However, you must enable the SERVO**STAR** in order to see motion:

```
Enable = ON
```

CAMming cannot be enabled if the axis is in a relative/absolute move or during stop. Enabling CAMming when gearing is enabled or when an absolute move is being executed will generate an error. Disabling or enabling the SERVO**STAR** SC does not effect whether gearing is enabled.

To disable CAMming, set SLAVE = OFF:

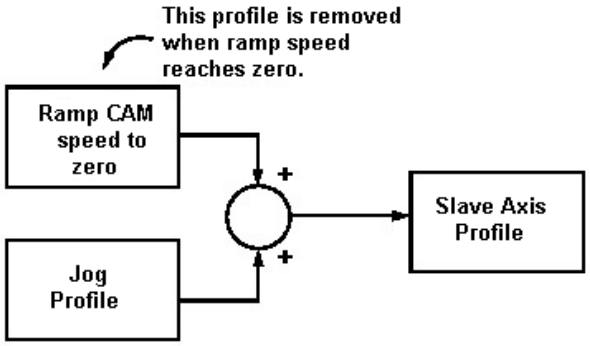
```
Slave  = OFF            'Disable CAMming
```

When CAMming is disabled, the velocity command of the slave axis is decelerated to zero at the rate of the axis property, DECSTOP.

Also, issuing a JOG or STOP command for the axis will disable CAMming.

Issuing a STOP command turns CAMming off immediately. The Velocity command of the axis is decelerated to zero at the rate set by the axis property, DECSTOP.

If a JOG command is issued, CAMming on that axis is disabled immediately, and the CAM profile ramps to zero at the rate DEC. However, the JOG profile ramps up at the rate specified by ACC. As with gearing, for a short time, the axis profile is the sum of these two profiles as shown in the next figure.



As with gearing, after the CAMming profile goes to zero, the profile is controlled wholly by the JOG profile. Disabling the master or slave does not disable CAMming. CAMming remains active and the slave position will be commanded accordingly when re-enabled. If the distance between the slave position and the command from the CAM table is greater then POSITIONERRORMAX when the SERVO**STAR** is re-enabled, a fault will be generated. Be cautious when re-enabling a slave axis that is CAMmed. If a large position error exists, the motor immediately corrects this error. The motor moves rapidly upon enable.

READ DYNAMIC INFORMATION

There are numerous read-only properties regarding the operation of the CAM.

AXIS PROPERTIES

ACTIVECAM is an axis property that provides the name of the CAM being executed by that axis.

CAM PROPERTIES

The CAM property, CASCYCLE, provides the number of cycles the CAM has executed. If the CAM master is moving forward, the current CAM ends at the completion of the cycle where CASCYCLE = CYCLE. When the CAM master is moving backward, the current CAM ends at the completion of the cycle where CASCYCLE = 0.

The CAM property, CAMINDEX, provides the current index within the current CAM table. For example, if there are 100 points in the table and the CAM is half complete, CAMINDEX is 50.

The CAM property, MASTERDATA, is used with CAMINDEX to provide the value of the master position of the current index within the CAM. For example, if the current CAM table master position range is -10 to +10 and the position is in the middle of the range, CAM1.MASTERDATA[CAMINDEX] is 0.00. CAM1.MASTERDATA[CAMINDEX] is not equivalent to the position of the master signal (PFB) because CAMs are incremental with respect to starting position.

SLAVEDATA can be used with CAMINDEX to give the value of the slave position for the current index of the CAM. As with CAMVALUE, CAM1.SLAVEDATA[CAMINDEX] is not equivalent to PCMD because the two may be offset from each other.

This page intentionally left blank.

INPUT/OUTPUT AND PLS

The SERVO**STAR** SC provides numerous types of I/O:

- 2 Dedicated Digital Inputs: Enable,Motion
- 3 Dedicated Inputs: Configurable to Home, Limit Switches, or Capture
- 1 Dedicated Digital Output: Motor Brake Control
- 19 Programmable Digital Inputs
- 10 Programmable Digital Outputs
- 2 Programmable Analog Inputs
- 2 Programmable Analog Outputs

General Purpose Digital I/O

The SERVO**STAR** SC includes a standard of 19 Inputs and 10 Outputs. These are accessed with the system properties DIN.2 through DIN.20 and DOUT.1 through DOUT.10:

```
? System.Din.1
System.Dout.5 = System.Din1
?System.Din
System.Dout=0xfe
```

For information concerning the electrical properties of the signals that connect to these inputs and outputs, refer to the various input and output descriptions as well as the *SERVO**STAR**® SC Installation Manual*.

Dedicated Digital I/O

The SERVO**STAR** SC has 2 dedicated inputs that must be connected for motion to occur. These are remote input on Connector C3 and motion input on connector C9. The status of these inputs can be checked by entering from the terminal "?REMOTE" and "?SYS.DIN.1".

The SERVO**STAR** SC has three other dedicated yet configurable digital inputs, (IN1, IN2, and IN3) and one brake output. The digital output (O1, O1MODE) is toggled in an On/Off state to indicate enable or disable and used for motor braking. The functionality of the three digital inputs can be home, limit switch, or capture is set using IN1MODE, IN2MODE, and IN3MODE.

For information concerning the properties of the signals that connect to these inputs and outputs, refer to the input and output descriptions as well as the *SERVO**STAR**® SC Installation Manual*.

Analog I/O

The SERVO**STAR** SC has two analog inputs, (ANIN1 and ANIN2) and two analog outputs (ANOUT1 and ANOUT2). These are configured with the system properties ANOUT1MODE and ANOUT2MODE. ANIN1 and ANIN2 are read using ANIN1MODE and ANIN2MODE.

For information concerning the properties of the signals that connect to these inputs and outputs, refer to the input and output descriptions as well as the *SERVO**STAR**® SC Installation Manual*.

PLS (Programmable Limit Switch)

A Programmable Limit Switch (PLS) monitors position feedback on the axis. A PLS offers more flexibility than a limit switch as it allows many trip points, and a PLS can be reconfigured on-the-fly. A PLS toggles the state of a specified system output when an axis position reaches any of the defined positions of the PLS. Some of the features of a PLS are:

- Multiple position specifications.
- Multiple PLSs per axis
- PLS pattern repetition.
- Position hysteresis
- Enabling and disabling of PLSs.
- The system output may be Digital or Soft (fast data)

PLS Positions are defined in an array, similar to the CAM data mechanism, which allows the user to change individual values of PLS positions; this array must be monotonic, increasing. There is no limit on the number to be defined or number of positions defined for the PLS.

The initial output polarity is specified via the PLSPOLARITY property, and the output state is set when the PLS is enabled. PLS properties may be changed only when the PLS is disabled.

Enable and Disable

A PLS can be enabled or disabled. After being initially defined, a PLS is automatically disabled. This means the output pattern is not generated. The output pattern is set when the PLS is enabled.

PLS properties can be changed only when the PLS is disabled. Each PLS that is enabled requires CPU resources, so you can conserve those resources by disabling PLSs that are not in current use.

The status of the PLS is controlled by the property PlsEnable:

```
MYPLS.PLSENABLE = ON | OFF
```

The PLSENABLE property is used to query the status of a PLS:

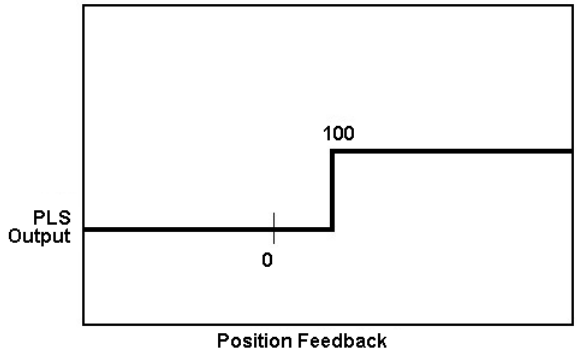
```
?MyPls.PlsEnable
if(MyPLS.PlsEnable = OFF)
```

or to drive events:

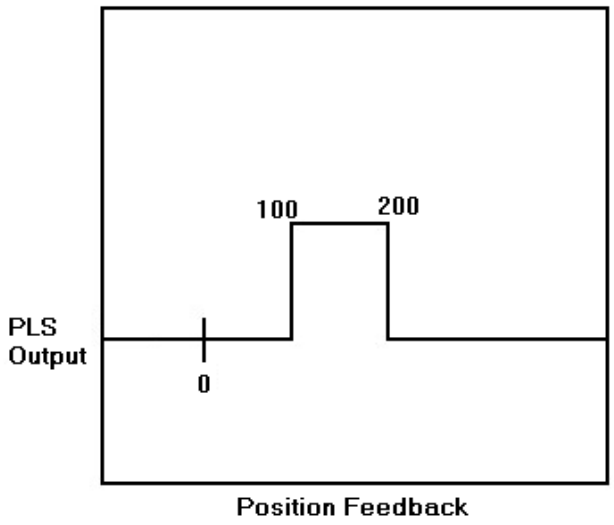
```
EventOn MyEvent MyPLS.PlsOutput = ON
```

Switch Positions

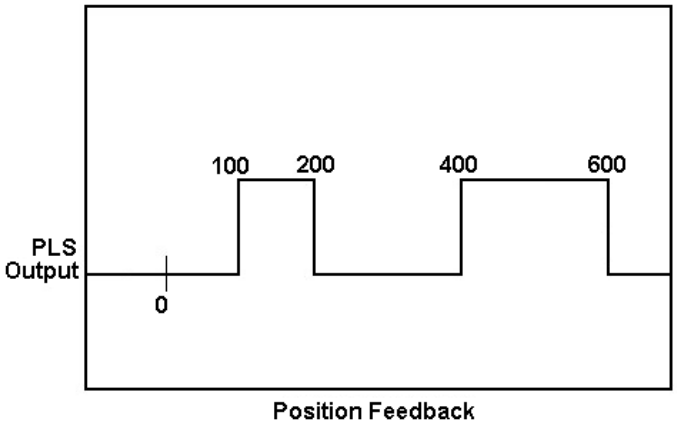
The basic mechanism of the PLS is the switch position or positions where the PLS output is to change state. The following figure shows a PLS output with one switch position at 100.



More switch positions can be added. To have the PLS in the previous example turn back off at 200, simply add a second position at 200 as shown below.



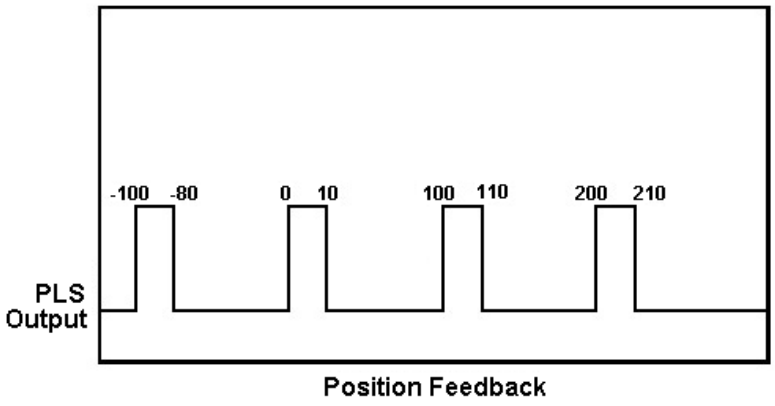
You can add any number of positions. For example, we can extend the above example to turn back on at 400 and off again at 600 by adding positions 3 and 4 as 400 and 600 as shown below.



There is no explicit limit to the number of positions you can have in a PLS., but the PLS positions must be monotonic increasing. That is, POSITION2 must be greater than POSITION1, POSITION3 must be greater than POSITION2, etc.

Repetition Interval

The PLSREPEAT property of a PLS allows you to repeat the same PLS after a fixed number of counts. For example, the next figure shows a PLS that turns on at 0, off at 10 and has a PLSREPEAT value of 100.



Polarity

The PLS property, PLSPOLARITY, allows you to control the polarity of the PLS output. The initial polarity may be specified as 1 or 0, indicating that the output will be set to 1 or 0 after the PLS is enabled and when the axis is located between the first and second PLS positions. If the axis position when the PLS is enabled is not located within this segment, the output polarity will be set to 0 or 1 according to the current position so that when the axis reaches the first segment the output polarity will be as indicated. The default value for polarity is 1.

Hysteresis

The SERVO**STAR** SC allows the user to apply hysteresis around the switch points. This hysteresis provides margin in the switching between PLS positions to accommodate noise in the PLS. You should set the hysteresis greater than the noise on the axis, typically 5 or 10 counts of encoder is sufficient; less is required for resolver systems. If you have a hysteresis of, say 0.01 position units, each transition of PLS position will be moved forward 0.005 units for positive motion and backward 0.005 units for negative motion. That way, if the axis driving the PLS is stopped directly on a PLS position, and the inherent noise on the axis is less than the hysteresis, the PLS state will not change due to this noise.

PLS Output State

To query the state of the PLS output, you must query the digital output that is associated with the PLS. For example, assume the PLS output is assigned to SYS.DOUT.1; then, to determine the output state of the PLS, query the state of SYS.DOUT.1

```
-->?Sys.Dout.1
0 | 1
```

PLS Set Up

To setup a PLS, use the following steps:

Step 1: Common Shared

The first step to set up a PLS is to declare the PLS with the Common Shared command in Config.Prg, or at the terminal. You must name the PLS and name an output to be controlled by the PLS.

```
Common Shared MyPLS Sys.Dout.1
```

Sets up MyPLS as a PLS connected to SYS.DOUT.1. The output can be a digital output (SYS.DOUT.1 to SYS.DOUT.20) or it can be one of the virtual outputs (SYS.VOUT.1 to SYS.VOUT.32). After the Common Shared statement is executed, the PLS is disabled. Properties of the PLS are set as:

- A single PLS position exists at 0
- The initial output polarity is 1.
- PLSRepeat is set to 0.
- The hysteresis is set to 0.

These properties may be set explicitly, as long as the PLS remains disabled. You can check which output is associated with a given PLS by using the PLSOUPUT property.

Step 2: Set the PLS Positions
The next step is to create the PLS data structure and define the PLS Positions. The number of positions you can define is not explicitly limited. A call to CREATEPLSDATA creates an array of *n* positions that store the position data. The array is 1 based and an error will be generated if you attempt to access an index outside of the array bounds. For example:

```
CreatePLSData 4 MyPLS

MyPLS.PLSPosition[1] = 1000
MyPLS.PLSPosition[2] = 1100
```

As a short form, you can use PPOS in place of PLSPOSITION.

```
MyPLS.Ppos[3] = 2000
MyPLS.Ppos[4] = 2200
```

You cannot change PLS positions when the PLS is enabled, and position values must be monotonically increasing.

Step 3: Set the Polarity
Now, set the polarity. PLSPOLARITY defaults to ON, indicating the PLS state should be ON after the first position. Change PLSPOLARITY to OFF if you want to invert the state.

```
MyPLS.PLSPolarity = OFF
```

You can change PLSPOLARITY only if the PLS is disabled.

Step 4: Set the Repetition Interval
If you want the PLS to repeat, set the value of the PLS property PLSREPEAT to some non-zero positive number. For example, to cause the PLS to repeat every 10000 position units, enter:

```
MyPLS.PLSRepeat = 10000
```

PLSREPEAT defaults to 0, indicating that there is no repetition in the PLS. You can change PLSREPEAT only when the PLS is disabled.

Step 5: Set the Hysteresis Level
Set PLSHYSTERESIS if the application requires it. Hysteresis is only necessary if the system will stop on or near a PLS position. Normally, a PLSHYSTERESIS of 5 or 10 counts of encoder (converted to position units) or 2 or 3 counts of resolver resolution is sufficient.

```
MyPLS.Hysteresis = 0.01
```

Step 6: Enable the PLS
Now you are ready to enable the PLS. Enter

```
MyPLS.PLSEnable = ON
```

Changing Polarity (Optional)

The default output state of the PLS is 1. You can change this initial state by modifying the PLSPOLARITY property.

```
MyPLS.PLSPolarity = 0
```

The PLS must be disabled when changing this setting.

Disabling the PLS (Optional)

You can disable the PLS

```
MyPLS.PLSEnable = OFF
```

You must disable a PLS to change PLS properties. Disabling a PLS also helps conserve CPU resources.

Deleting the PLS (Optional)

You can delete a PLS to remove it from the system only when no tasks are using the PLS. The following is an example of removing a PLS:

```
DeletePLS MyPLS
```

MODBUS PROTOCOLS

The MODBUS communication protocols allow supervision and control of automation equipment. The MODBUS protocols are supported by subroutines called from a library. This document assumes that you have an understanding of MODBUS TCP/IP and MODBUS RTU protocols. This document can be used with the *Open MODBUS TCP Specification* and *MODBUS Protocol* documents available from MODICON Inc.(industrial automation systems). MODBUS protocols are only supported in the SERVO**STAR** SC in versions 2.1.9 and higher. For details about the MODBUS Library subroutines, refer to the *SERVO**STAR** SC Reference Manual*.

Slave

With MODBUS TCP/IP and MODBUS RTU library subroutines, the SERVO**STAR** SC acts as a Slave waiting for a query from the MODBUS Master and sends the appropriate response to it. Because there are two communication methods, there are two ways of connecting to the master.

Connecting via TCP/IP to the Master

When establishing a TCP/IP communication with a MODBUS Master, the SERVO**STAR** SC acts as the TCP/IP server, binding the socket to a specified port and waiting for the connection. To do this, you would type:

```
OPENSOCKET Options=0 As #1
Accept(#1,502)
```

Connecting via RTU (Serial) to the Master

When establishing an RTU or serial communication with a MODBUS Master, the SERVO**STAR** SC you must open the serial communication channel. To do this, you would type:

```
OPEN COM2 BaudRate=9600 Parity=0 DataBits=8 StopBit=1 As #1
```

Setting the Communication Method

The subroutine, MODBUSInitSlave designates the communication form for the library functions. The last variable sent is 2 for TCP/IP or 1 for RTU or serial communication. Communication must be established before calling the MODBUSInitSlave subroutine. The handle must also be passed to this subroutine. For example, if the connection was:

```
Accept(#1,502)
```

Initialize the MODBUS communication library with:

```
Call MODBUSInitSlave(1,2)
```

This command tells the SERVO**STAR** SC to work through handle #1 and to use communication type two (TCP/IP). If, on the other hand, the connection was:

```
OPEN COM2 BaudRate=9600 Parity=0 DataBits=8 StopBit=1 As #1
```

Initialize the MODBUS communication library with:

```
Call MODBUSInitSlave(1,1)
```

This command tells the SERVO**STAR** SC to work through handle #1 and to use communication type one (RTU or serial).

Changing Masters

Since the SERVO**STAR** SC can only be a slave for one master, if the communication needs to be changed from TCP/IP to RTU or serial (or vice versa), the MODBUSStopSlave subroutine must be called first. After the call to this subroutine, the subroutines MODBUS InitSlave and MODBUSSlave must both be called to start the new communication form. If another master tries to call the MODBUSSlave subroutine while the SERVO**STAR** SC is working as a slave, nothing happens and the SERVO**STAR** SC continues to work for the former master.

Supported Functions

The supported MODBUS functions are:

FC 0x01 – Read Multiple Coils
FC 0x02 – Read Multiple Inputs
FC 0x03 – Read Registers
FC 0x04 – Read Input Registers
FC 0x05 – Write Single Coil
FC 0x06 – Write Single Register
FC 0x0F – Write Multiple Coils
FC 0x10 – Write Multiple Registers
FC 0x11 – Read SC data

FC 0x11 Returned Data

Function code 17 (0x11 HEXA) returns the data (only in data bytes) in a form particular to the SC as slave:

Byte 1: 13 (bytes following)
Byte 2: devise address (defaults to 1 if not changed)
Byte 3: 1 (means that the slave ID is in the data)
Byte 4: 1 (the indicator is ON)
Byte 5: high byte of the serial number (ID)
Byte 6: low byte of the serial number (ID)
Byte 7: amount of coils possible to use high byte
Byte 8: amount of coils possible to use low byte
Byte 9: amount of registers high byte
Byte 10: amount of registers low byte
Byte 11: amount of input bits possible to use high byte
Byte 12: amount of input bits possible to use low byte
Byte 13: amount of input registers high byte
Byte 14: amount of input registers low byte

Register Data Flow

Data returned from the MODBUS slave containing registers FC 0x03, 0x04, and 0x10 are returned in two bytes representing the value of a 16-bit register. For example, if the returned register's value is 437, the returned value is 1 for the high byte and 181 for the low byte of the register. To get the actual value, it would be:

0x100*(high byte)+low byte.

Single Bit Data Flow

Data returned from the MODBUS slave containing single bits (FC 0x01 and 0x02) are returned in bytes representing 8 bits, starting from the first bit to read. The bit addresses start from address 0 for bit 1.

Data Reference

The reference to coils or input bits is always one lower than the bit in the slave. The reference to the register is the same. For example, to reference coil 1, the reference in the master query should be 0. To reference register 5, the reference in the master query should be 4.

Inputs and Outputs

There are four library subroutines to control the inputs and outputs from outside the master. Two are for setting inputs and outputs, while two more get the values of the inputs and outputs. For additional information, refer to the *SERVO**STAR** SC Reference Manual*.

MODBUSSETBIT

This library subroutine sets the value of a single bit to ON or OFF. The bit to be changed is in the output data if the last function variable is 1, and in the input data if the last function variable is 2. The data starts from one bit lower than the real memory reference. So bit 1 is referenced as 0. For example:

```
Call MODBUSSetBit(4,1,1)
```

Sets coil number 5 to 1 while:

```
Call MODBUSSetBit(3,0,2)
```

Sets input bit number 4 to 0.

MODBUSSETREGISTER

This library subroutine sets the value of a register. The register to be changed is in the output data if the last function variable is 1, and in the input data if the last function variable is 2. The data starts from one register lower than the real memory reference. So register 1 is referenced as 0. For example:

```
Call MODBUSSetRegister(4,100,1)
```

Sets input register number 5 to 100 while:

```
Call MODBUSSetRegister(13,23,1)
```

Sets register number 14 to 23.

MODBUSGETBIT

This library subroutine gets the value of a single bits. The status of the bit is from the output data if the last function variable is 1, and from the input data if the last function variable is 2. The data starts from one bit lower than the real memory reference. So register 1 is referenced as 0. For example:

```
? MODBUSGetBit(8,1)
```

Returns the status of the 9th coil while:

```
? MODBUSGetBit(1022,2)
```

Returns the status of the 1023rd input bit.

MODBUSGETREGISTER

This library subroutine gets the value of a register. The status of the register in the output data if the last function variable is 1, and in the input data if the last function variable is 2. The data starts from one register lower than the real memory reference. So register 1 is referenced as 0. For example:

```
? MODBUSGetRegister(8,1)
```

Returns the value of the 9th output register while:

```
? MODBUSGetRegister(1022,2)
```

Returns the value of the 1023rd input register.

MODBUS TCP/IP Slave Example

1. Establish a connection to a remote host to send and retrieve characters:

```
OPENSOCKET Options=0 As #1
Connect (#1,502)
```

2. Initialize the MODBUS communication library:

```
Call MODBUSInitSlave(1,2)
```

3. Get the slave to scan the communication handle for queries:

```
Call MODBUSSlave
```

4. To stop the function MODBUSSlave from scanning the handle or to reset all the variables in order to change the master:

```
Call MODBUSStopSlave
```

5. To close the communication channel:

```
Close #1
```

MODBUS RTU Slave Example

1. Open the serial communication channel:

```
Open COM2 BaudRate=9600 Parity=0 DataBits=8 StopBit=1 as #1
```

2. Initialize the MODBUS communication library:

```
Call MODBUSInitSlave(1,2)
```

3. Get the slave to scan the communication handle for queries:

```
Call MODBUSSlave
```

4. To close the communication channel:

```
Close #1
```

Master

When establishing communication with a MODBUS slave device, the SERVO**STAR** SC acts as the CLIENT, requesting a connection from a remote host according to the server's IP address and port (the slave's IP address and port). In order to choose the form of connection (TCP/IP or RTU), the subroutine MODBUSINIT must be used. The last variable sent will be 2 for TCP/IP or 1 for RTU. For a TCP/IP connection with slave device number 1 at handle #1 of communication type 2 (TCP/IP), you would type:

```
OPENSOCKET Options=0 As #1
CONNECT (#1,"212.25.84.99", 502
Call MODBUSINIT (1,1,2)
```

For an RTU or serial connection with slave device number 1 at handle #1 of communication type 1 (RTU or serial), you would type:

```
OPEN COM2 BaudRate=9600 Parity=0 DataBits=8 StopBit=1 As #1
Call MODBUSINIT (1,1,1)
```

Unlike normal MODBUS TCP/IP and MODBUS RTU protocols, the SERVO**STAR** SC subroutines that deal with MODBUS do not require any framing for the messages, since the frames are encapsulated within the SC subroutines and only the data itself is sent to the subroutines. For example, in normal MODBUS communication, for the function coded as 0x1 (read 1coil at reference 21) you would need to send:

```
00 00 00 00 00 06 01 01 00 20 00 01
```

However, using SC subroutines, only Device address=1, starting address=21, and number of points=1 needs to be sent. The call for the subroutine would look like:

```
Call MODBUSReadCoil(1,20,1,RESPONSE_ARRAY)
```

The data returned from the subroutine is in the RESPONSE_ARRAY.

Memory Reference

The memory reference in the MODBUS is the first coil, input, coil register or input register is referenced from 0. However, the read address starts from 1 so the referencing is always one less than the actual address. For example, data is referenced from the fifth address because the reference number sent by the subroutine is 4:

```
Call MODBUSWriteMultipleRegisters(17,4,2,DATA_ARRAY)
```

Register Data Flow

Data passed to and from the MODBUS subroutine's registers (FC 0x3, 0x4, 0x10) is sent in complete words. When sending and receiving 16-bit values of data, use long variables to hold the data. For example:

```
Call MODBUSReadInputRegisters(17,9,4,RESPONSE_ARRAY)
```

Requests the value of four 16-bit registers from device number 17 starting from offset 10. The returned data is sent back by the subroutine in four words. The size of the RESPONSE_ARRAY should be four elements, each of one word (two bytes). If you wish to write to multiple registers, consider this example:

```
Common shared DATA_ARRAY[2] as long     'an array of 2 to write values into 2
                                        'registers of 16-bits each
DATA_ARRAY[1]=0xF00
DATA_ARRAY[2]=0x24
MODBUSWriteMultipleRegisters(17,2,2,DATA_ARRAY)
```

This example writes two registers (each 16 bits) at offset 3 of values 0xF00 and 0x24.

Single Bit Data Flow

To send or receive single bits, the data needs to be arranged in a single byte for each 8 bits. For example, to read coils 21 through 36 from slave device 17, an array of 2 must be defined:

```
Common shared RESPONSE_ARRAY[2] as long
```

Then, the subroutine can be called:

```
Call MODBUSReadCoil(17,20,16,RESPONSE_ARRAY)
```

The returned RESPONSE_ARRAY[1] will contain the data of bits 21 through 28. RESPONSE_ARRAY[2] will contain the data of bits 29 through 36. If, for example, bit 21 is 1, bit 22 is 1 and bits 23 through 28 are 0. RESPONSE_ARRAY[1] returns 3.

To write coils 21 through 26 in slave device 1, you would enter:

```
Common shared DATA_ARRAY[1] as long
DATA_ARRAY[1]=11
Call MODBUSWriteMultipleRegisters(1,20,6,DATA_ARRAY)
```

In this example, coils 21, 22, and 24 are turned on and coils 23, 25, and 26 are 0.

MODBUS TCP/IP Master Example

1. Configure the MODBUS slave device according to the device's manual.
2. Connect to the slave with IP address 212.25.84.99 and port 502:

```
OPENSOCKET Options=0 As #1
Connect (#1,"212.25.84.99",502)
```

3. Initialize the MODBUS communication library:

```
Call MODBUSInit(1,1,2)
```

4. Prepare an array:

```
Common shared DATA_ARRAY[1] as long
DATA_ARRAY[1]=11
```

5. Call the subroutine (in this example, turning bits 31, 32, and 34 on and bits 33, 35, and 36 off):

```
Call MODBUSWriteMultipleRegisters(1,30,6,DATA_ARRAY)
```

6. To close the communication channel:

```
Close #1
```

This entire example would be written as:

```
OPENSOCKET Options=0 As #1
Connect (#1,"212.25.84.99",502)

Call MODBUSInit(1,1,2)

Common shared DATA_ARRAY[1] as long
DATA_ARRAY[1]=11

Call MODBUSWriteMultipleRegisters(1,30,6,DATA_ARRAY)

Close #1
```

MODBUS RTU Master Example

1. Open the serial communication channel:

```
Open COM2 BaudRate=9600 Parity=0 DataBits=8 StopBit=1 as #1
```

2. Initialize the MODBUS communication library:

```
Call MODBUSInit(1,1,2)
```

3. Prepare an array:

```
Common shared DATA_ARRAY[1] as long
DATA_ARRAY[1]=11
```

4. Call the subroutine (in this example, turning bits 30, 31, and 33 on and bits 32, 34, and 35 off):

```
Call MODBUSWriteMultipleRegisters(1,30,6,DATA_ARRAY)
```

5. To close the communication channel:

```
Close #1
```

This entire example would be written as:

```
Open COM2 BaudRate=9600 Parity=0 DataBits=8 StopBit=1 as #1
Call MODBUSInit(1,1,1)

Common shared DATA_ARRAY[1] as long
DATA_ARRAY[1]=11

Call MODBUSWriteMultipleRegisters(1,30,6,DATA_ARRAY)

Close #1
```

This page intentionally left blank.

TROUBLESHOOTING

Technical papers and publications about the SERVO**STAR** and its associated applications complete the information package necessary for the user to become well versed with the product. Kollmorgen's engineering and technical resource staff has prepared these notes. Also included are the *SERVO**STAR**® SC Setup Guide* and the *SERVO**STAR**® SC Installation Manual*. The *SERVO**STAR**® SC Installation Manual*'s Troubleshooting section provides error codes and an explanation of the troubleshooting tools used in BASIC Moves Development Studio, **MOTIONLINK**, and MotionSuite. The most recent versions of all the material contained in this PSP CD-ROM can be downloaded from Kollmorgen's website (http://www.MotionVillage.com).

Customer Support

Kollmorgen is committed to quality customer service. Our goal is to provide the customer with information and resources as soon as they are needed. In order to serve in the most effective way, please contact your local sales representative for assistance. If you are unaware of your local sales representative, please contact us at:

Danaher Customer Support - Radford
501 West Main Street
Radford, VA 24141
Phone: 1-800-777-3786 or (540) 639-9046
Fax: (540) 639-1574
Email: servo@kollmorgen.com

Visit our web site at www.MotionVillage.com for software upgrades, application notes, technical publications, and the most resent version of our product manuals.

This page intentionally left blank.

APPENDIX A

Digital Incremental Encoder Types

The SERVO**STAR** SC products include models designed for use with incremental digital encoders (SC1E). Encoders are available in different configurations from a variety of manufacturers using different nomenclature and conventions, making this topic somewhat confusing. Encoders can be as simple as having only A and B output channels or as complex as 6 channels of outputs. The SERVO**STAR** supports many of these variants using the variable, MENCTYPE. The variable is set according to the features of the encoder. This document explains the different applications of the various MENCTYPEs. A basic understanding of encoders is valuable for this discussion.

Encoder Basics: A Review

Encoders used with the SERVO**STAR** provide incremental motor position information via two channels, referred to as the A Channel and B Channel. These channels output pulses for a unit of shaft motion. These pulses are typically generated within the encoder, using an optical disk that is directly connected to the motor shaft. The disk has etchings that either transmit or block light passing through the disk. An optical transmitter and receiver are on either side of the disk. The rotation of the disc (and motor shaft) interrupts light transmission from source to receiver, creating the pulses. The interruptions on the disk are called lines and result in the encoder ratings of lines-per-revolution (LPR) or pulses-per-revolution (PPR). The two channels provide the same information (pulses-per-unit of motion), but have a phase shift of 90º between each other (See Figure A - 1). The 90º electrical phase shift between the two channels is referred to as "quadrature-encoded." The encoder output appears as a frequency, but the pulse rate is dependent on the motor's rotational velocity, not time.

Since the two channels are phase-shifted by 90º, there are actually four states available per electrical cycle of these signals (see Figure A - 1). The SERVO**STAR** is able to receive four counts for position feedback for one line of motion of the encoder. The actual decode of the four position counts-per-line of the encoder is called "quadrature decode." Additionally, since the encoder signals A and B are phase shifted by 90º, it is easy to design electronics that recognize whether A came before B or B came before A, thus supplying directional information.

Encoders are often provided with an additional channel called a "Marker" channel, Zero Pulse, or an "Index" channel - different names for the same function. This channel outputs one pulse per revolution and is typically an extremely narrow pulse equating to roughly ¼ of the width of an A or B channel pulse but can be wider. This is a reference position marker used for homing (absolute position reference) and commutation alignment.

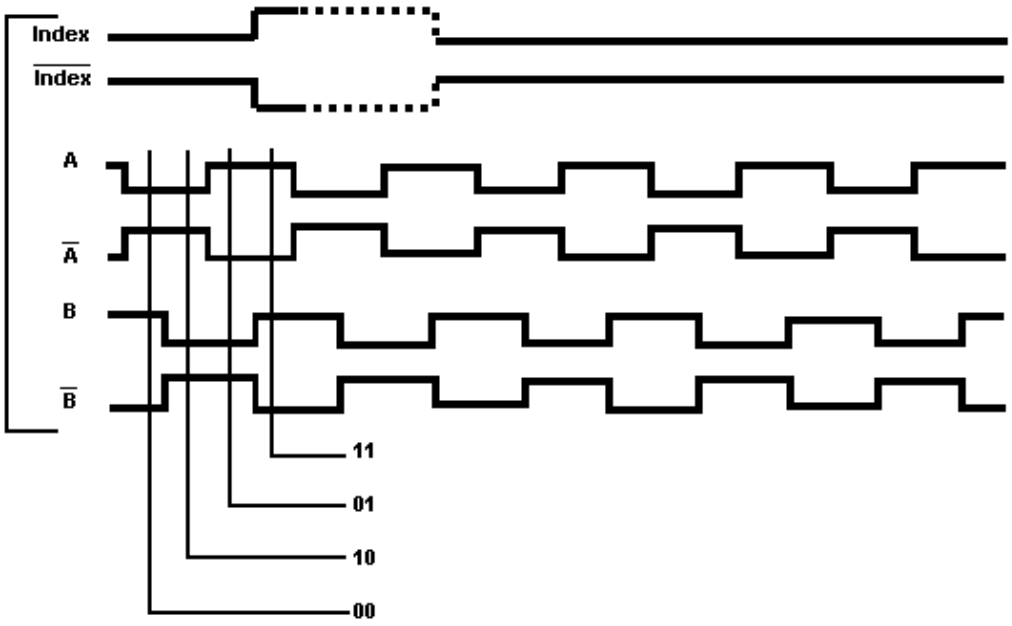


Figure A - 1: Basic Encoder Signals (Differential)

One challenge when using encoders is that they are incremental rather than absolute devices. When using an absolute device (such as a resolver) to determine the motor shaft position within the rotation, the transmitted code is unambiguous. In other words, at power-up, the system knows the position of the motor shaft.

Incremental encoders only detects how far the motor shaft has moved from its original position. This presents a problem with three-phase brushless motors in terms of commutation alignment. Generally, it is extremely important to establish the appropriate commutation angle within the controller. (Commutation refers to the alignment of the electromagnetic field armature winding to the permanent magnet fields to create optimal torque.)

For this reason, encoders or motors are often provided with additional channels sometimes called "commutation tracks" or "Hall emulation tracks" which provide 1-part-in-6 absolute position information (See Figure A - 2). The hall channels can be synthesized in the encoder or can be discrete devices integrated in the motor windings. Commutation tracks (hall channels) provide three digital channels that represent alignment to the A-phase, B-phase, and C-phase back EMF of the motor. An encoder with Hall channels must have the correct output for the given pole count of the motor as the Hall signals are referencing the motor's BEMF waveform.

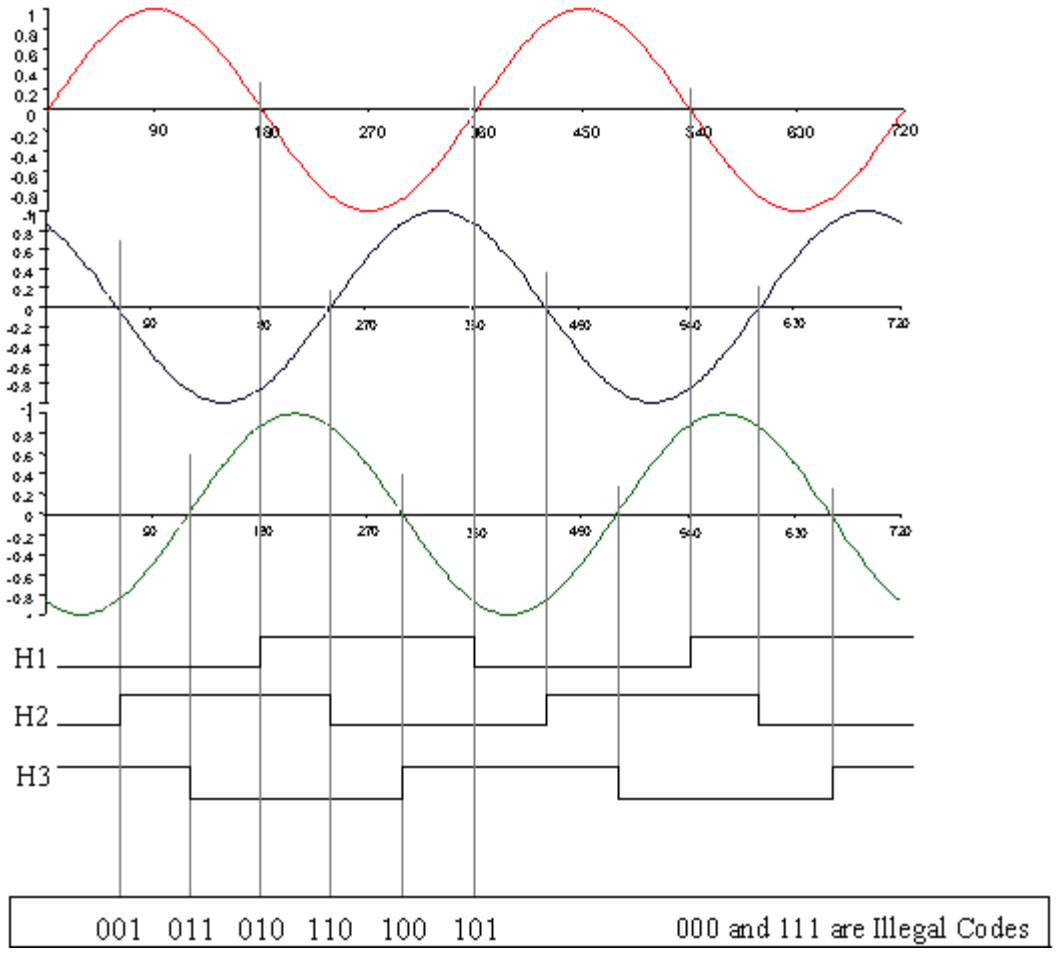


Figure A - 2: Hall Channel Signals

Some systems use only hall channels for motor feedback data. The channels provide enough information to commutate a motor in an ON and OFF (trapezoidal) format but do not provide enough information to properly commutate a motor using sinusoidal control. Further, the coarse data is insufficient to control velocity below a few hundred RPM. The SERVOSTAR product is a high-performance controller and does not support hall-only operation.

Since encoders are incremental devices, a loss or gain of a pulse creates system errors. Electrical noise is the single biggest factor in miss-counts. Transmitting the signals differentially provides the largest margin of noise rejection and the best signal fidelity. The SERVO**STAR** is designed to receive only differential signals. Some "less expensive" encoders provide TTL or "Open Collector" signals. These are not compatible with the SERVO**STAR**.

SERVOSTAR Encoder Types

The idea of obtaining velocity or position information from the series of pulses generated from the encoder is not difficult to understand. Permanent magnet brushless DC servo motors require commutation. As stated earlier, commutation is simply the positioning of the electromagnetic fields in alignment with the permanent magnet fields such that optimal torque is produced. This requires that the motor shaft position be known at all times. The use of incremental encoders requires some form of initialization to determine this motor shaft position at power up. It is this commutation initialization process that can lead to confusion.

Since the SERVO**STAR** supports operation with many encoder types, it must be told which signals to expect to see and which initialization algorithm to perform. This is the purpose of the MENCTYPE variable. The following section explains the MENCTYPE variable and its setting for each encoder type. Figure A - 3 shows the initialization flow chart for the various MENCTYPEs.

MENCTYPE 0

Incremental with A/B/I and Hall Channels MENCTYPE 0 is the most robust encoder system. It is used to tell the drive to expect the signals as feedback from the motor. The hall channels can be synthesized in the encoder or can be discrete devices (Hall sensors) integrated in the motor windings. On power-up, the hall effect channels are read and a code is sent to the microprocessor to give it a coarse position for the motor. This position is accurate to within ±30 electrical degrees of the optimal commutation angle. The SERVO**STAR** assumes that the actual motor position is half-way between the hall code settings for a maximum commutation error of ±30 electrical degrees. Even with this amount of error the motor is still capable of producing torque with an efficiency of 86%. As the motor rotates, the first hall boundaries quickly traverse, providing the SERVO**STAR** with enough information to better establish commutation angle. The SERVO**STAR** further corrects the commutation angle after it sees the marker pulse according to the value of MECNOFF.

MENCTYPE 1

Encoders with A/B/I channels Some systems do not have hall channels, so this mode tells the SERVO**STAR** not to expect them. The initialization process occurs in two steps. The first step is the 'wake-and-shake initialization process (see MENCTYPE 3 and MENCTYPE 4 for "wake and shake" details) which gets the commutation alignment initialized after power up. The second step occurs when the index pulse is seen. Then, the SERVO**STAR** aligns the commutation angle according to the setting of MECNOFF. MENCTYPE 1 initialization begins only when commanded through the serial port by using the ENCSTART command.

The 'wake and shake' initialization process is not required if the marker location can be traversed without requiring the SERVOSTAR to move the motor to traverse the marker.

MENCTYPE 2

Encoders with A/B/I channels Some systems do not have hall channels, so this mode tells the SERVO**STAR** not to expect them. The initialization process occurs in two steps. The first step is the 'wake-and-shake initialization process (see MENCTYPE 3 and MENCTYPE 4 for "wake and shake" details) which gets the commutation alignment initialized after power up. The second step occurs when the index pulse is seen. Then, the SERVO**STAR** aligns the commutation angle according to the setting of MECNOFF. MENCTYPE 2 is initialized using ENCSTART or performed automatically upon power-up when the drive is enabled.

The 'wake and shake' initialization process is not required if the marker location can be traversed without requiring the SERVOSTAR to move the motor to traverse the marker.

MENCTYPE 3

Encoders with A and B channels only The simplest of encoders provide only an A and B Channel. MENCTYPE 3 configures the SERVO**STAR** for these signals. These encoders provide no power-up information about where the motor is positioned, so this information is obtained through a special initialization process known as 'wake and shake'. When using MENCTYPE 3, initialization is required but it is triggered by the serial command, ENCSTART.

During the initialization process, "wake and shake," the SERVO**STAR** puts current through two phases of the motor causing the motor shaft to rotate into a "torque detent." The magnets simply align the motor shaft to a position the SERVO**STAR** understands. The motor is then rotated to the next pole position by energizing the windings in a slightly different manner. The SERVO**STAR** then has enough information to establish an appropriate commutation angle using the number of motor poles (MPOLES) and the number of counts for the encoder (MENCRES).

This method of initialization has the advantage of allowing an extremely cost-effective feedback device. However, the requirement to pull the motor into these torque detents is subject to outside influences (such as friction) and can prevent the motor motion from occurring very effectively or even at all! Additionally, large inertia loads can overshoot and oscillate during this pull-in position, giving the microprocessor false readings. These MENCTYPEs are best for systems with specific criteria requirements such as low friction, low cost, and low inertial loads. The amount of electrical current used in this initialization mode is adjustable using the IENCSTART variable and may need to be adjusted to optimize performance for large inertial loads.

MENCTYPE 4

Encoders with A and B channels only The simplest of encoders provide only an A and B Channel. MENCTYPE 4 configures the SERVO**STAR** for these signals. These encoders provide no power-up information about where the motor is positioned, so this information is obtained through a special initialization process known as 'wake and shake'. When using MENCTYPE 4, the process for initialization is automatically triggered on power-up when the drive is enabled or by using ENCSTART.

During the initialization process, "wake and shake," the SERVO**STAR** puts current through two phases of the motor causing the motor shaft to rotate into a "torque detent." The magnets simply align the motor shaft to a position the SERVO**STAR** understands. The motor is then rotated to the next pole position by energizing the windings in a slightly different manner. The SERVO**STAR** then has enough information to establish an appropriate commutation angle using the number of motor poles (MPOLES) and the number of counts for the encoder (MENCRES).

This method of initialization has the advantage of allowing an extremely cost-effective feedback device. However, the requirement to pull the motor into these torque detents is subject to outside influences (such as friction) and can prevent the motor motion from occurring very effectively or even at all! Additionally, large inertia loads can overshoot and oscillate during this pull-in position, giving the microprocessor false readings. These MENCTYPEs are best for systems with specific criteria requirements such as low friction, low cost, and low inertial loads. The amount of electrical current used in this initialization mode is adjustable using the IENCSTART variable and may need to be adjusted to optimize performance for large inertial loads.

MENCTYPE 5

Not supported

MENCTYPE 6

Incremental with A/B and Hall Channels MENCTYPE 6 supports encoders as described in MENCTYPE 0, but lacking the marker or index channel. This device is selected using MENCTYPE 6 and follows the same process used in MENCTYPE 0 with the exception of the final search for the marker pulse. The system establishes the commutation angle based on the Hall effect edge. MENCTYPE 6 is most commonly used with linear motors.

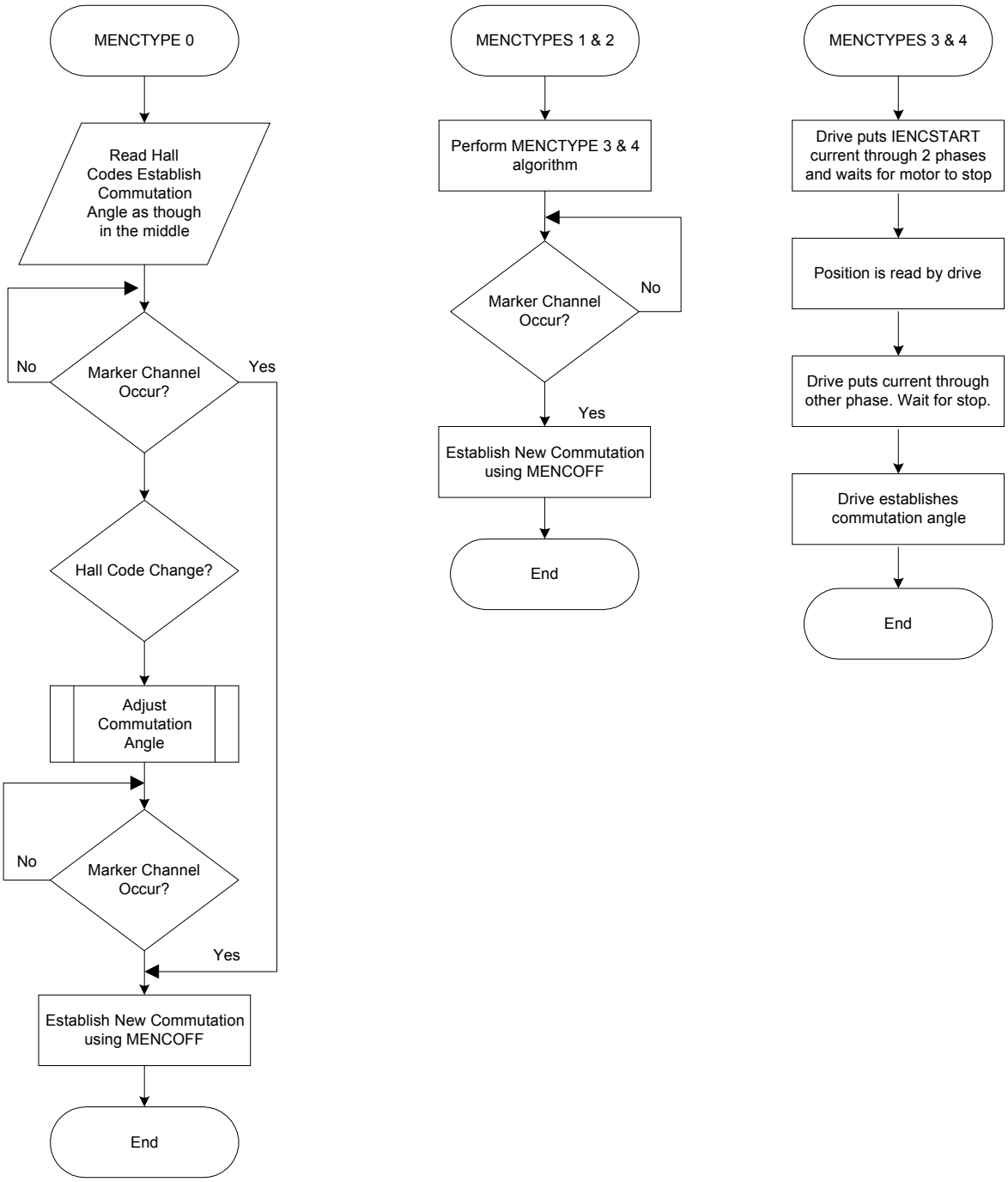


Figure A - 3: Initialization Flow Charts for Incremental Encoders

Commutation Accuracy

The accuracy of the commutation alignment within the drive affects the overall system efficiency. Misalignment also causes greater torque ripple. As a general estimator, the following equation holds:

$$K_{t\ effective} = K_{t\ Rated} * Cosine\ (Alignment\ Error)$$

Inaccuracy of commutation alignment can occur from multiple sources when using encoders. The accuracy of the hall channel devices, if they are provided from an encoder using an optical disc, are typically accurate to 5º or better electrically. When they are integral to the motor, they may have an error of up to 15º. Using the above equation, a 15º error would still provide more than 96% of the motor's torque.

The accuracy of the 'wake and shake' algorithms used in MENCTYPE 1, MENCTYPE 2, MENCTYPE 3, and MENCTYPE 4 are subject to many outside influences, such as friction and inertia. These algorithms are not robust and are subject to varying amounts of error.

Incorrect commutation alignment also causes differences in efficiency according to direction. In an exaggerated example, a misaligned commutation angle may allow a 1500 RPM motor to go 2000 RPM in one direction and only 1000 RPM in the other.

Physical Encoder Alignment

The SERVO**STAR** provides a number of features (variables) to allow adjustment of the encoder signal alignment through the software. These features can be very useful, but you must be careful when using them. If the alignment of the encoder to the motor is left to chance, swapping out the motor or encoder requires that these variables be recalibrated for the new alignment. Is this acceptable in the given application? You may not be certain that the knowledge to do so will remain in the future. A better method is to align the encoder on the motor. This makes swap-out a simpler process.

MENCTYPE 0 (A/B/I and Halls) has hall channels aligned to the motors back EMF as indicated in Figure A - 2. This is accomplished by rotating the motor via an external source, monitoring the signals from the hall effect channels while monitoring the voltage generated by the motor and rotating the encoder housing until the waveforms overlap. The encoder mounting screws are then secured. This aligns the hall channels and the marker channel must then be aligned using the MECNOFF variable. Caution must be exercised as motors may generate lethal voltages when rotating.

MENCTYPE 1 and MENCTYPE 2 (A/B/I without Halls) require the marker to be aligned to the motor's BEMF waveform. Kollmorgen has not defined any particular alignment standard. Defining your own alignment standard could be beneficial.

MENCTYPE 3 and MENCTYPE 4 (A/B only) require no alignment or adjustments.

MENCTYPE 6 (A/B/I with Halls) requires the same alignment as MENCTYPE 0 but does not require that MENCOFF be set.

When it is not practical to perform mechanical alignments, the software adjustment method can be used. The MPHASE variable holds an offset for the Hall channels and can be used to effectively 'shift' the Hall channel position. It is possible to invert the effective direction as established by the A and B channels using the MFBDR variable. It is also possible to invert any one or all of the hall channels. The MHINVA, MHINVB, MHINVC variables allow the drive to receive the hall channels and act upon them from an inverted nature.

MECNOFF

The MENCOFF variable holds a marker offset and is used to align the commutation in MENCTYPE 0, MENCTYPE 1, and MENCTYPE 2.

To determine the setting for MENCOFF perform the following steps using the **MOTIONLINK**® terminal mode with power on the drive and C3 unplugged (disabled):

1. Enter 'ENCINIT'.
2. Rotate the motor shaft two turns clockwise by hand.
3. Verify that the process is complete by entering 'ENCINITST'. The SERVO**STAR** should return '2' if the process has been successfully completed.
4. If not, repeat steps 2 and 3.
5. Enter 'SAVE'.
6. Use caution to continue testing the system.
7. The MENCOFF variable may be manually trimmed for best performance.

When using MENCOFF with MENCTYPEs 1 and 2, the correct value must be determined through trial and error. Pick a setting and trim it, watching for the speed (V) to be equal in both directions when applying torque in OPMODE 2.

It is common for repeated tries of this procedure to return values that are significantly different due to the software's reference point being different from try-to-try. This is normal.

The motor shaft must be free to rotate uncontrolled without damage to equipment or personnel.

System Phasing

With so many signals coming from the encoder back to the drive and then the signals going to the motor, it can become quite frustrating to make sure that each signal is of the appropriate phase. Figure A - 4 shows the phase relationship to each device.

This diagram shows the commutation phasing (PFB counts down) for:
Motors with shafts, counter-clockwise rotation viewing shaft end.
Motors without shafts, clockwise rotation viewing lead exit end.
Linear motors, with armature moving away from leads.

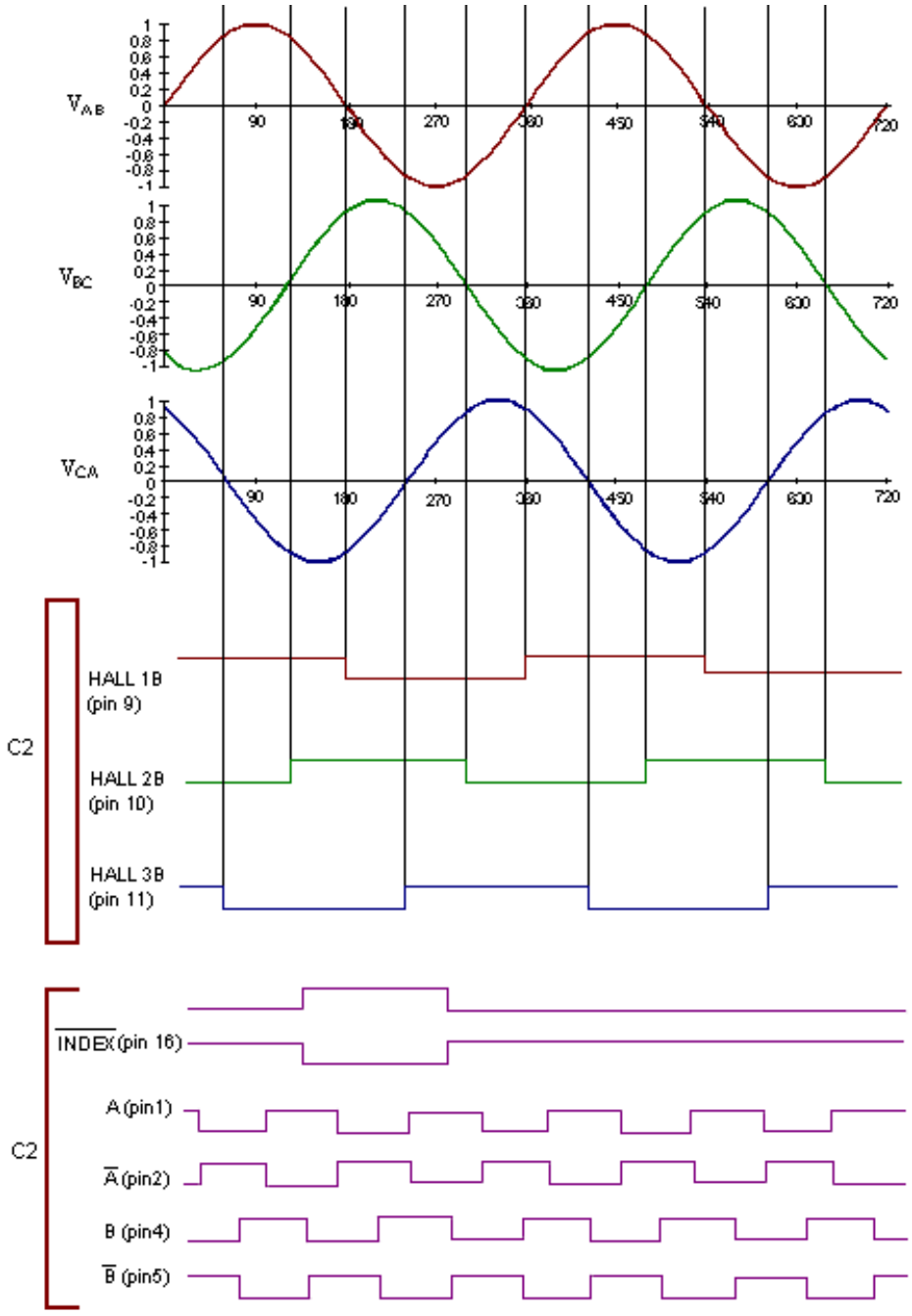


Figure A - 4: System Phasing Diagram

Troubleshooting

Several problems can occur with encoder-based systems. The most common problem is miswiring. The section on system phasing (above), provides the necessary troubleshooting information.

Miswired Hall channels can cause intermittent problems. With miswired Hall channels, the motor operates correctly sometimes, but will occasionally not operate correctly after cycling power. It is very important to physically verify the Hall effect channels. There is a command in the SERVO**STAR** manual called, "HALLS." It returns the hall code as "read." It is important that the installation and startup procedures for the machine sequence the motors through all the appropriate hall codes to make sure they are present and in the right sequence, as provided in Figure A - 2.

An 'illegal hall code' error occurs if the Hall channels go to all low or all high conditions. A broken wire or misphased channel can cause this problem.

It is important that the A and the B channels be wired appropriately to the SERVO**STAR** drive to ascertain the correct directional drive information. This can be verified by displaying the PFB variable while rotating the motor shaft clockwise. The PFB variable should be counting in a more positive direction. If it counts in a negative direction, the A and B channels are inverted. This can be fixed by swapping the A and /A wires from the encoder or by using the MFDIR command.

A/B Line Break errors may occur and can be misleading. The SERVO**STAR** receives the A/B/I channels in a differential format. Each channel is fed into a bridge rectifier to create a DC voltage that is monitored for presence. Absence of any one of these three voltages (except the I channel in MENCTYPE 3, MENCTYPE 4, and MENCTYPE 6) cause an A/B line break fault.

LINE DRIVERS, RECEIVERS, AND TERMINATIONS

Counting pulses sent over cables going through an industrial environment requires that care be taken to prevent noise induction on the cable that looks like an encoder pulse. Running wires in a "clean" raceway is one requirement.

Another common sense approach is to use differential transmission for the signals to provide the highest degree of noise immunity. Differential line drivers are required by the SERVO**STAR**. The differential line signals follow the RS-485 format where pulses are sent up and down a 120Ω cable. Termination is expected at both ends and the SERVO**STAR** provides the required termination. Deviations from a 120Ω characteristic impedance cable when using long cables can result in poor performance.

Design Considerations

The SERVO**STAR** has a maximum frequency input for incremental encoders of 2.5 MHz. This cannot be exceeded under any circumstance.

Encoders consume 5V. The 5V is typically regulated inside the drive through a cable to the encoder. Tolerances on the 5V must be considered for IR loss within the cable. An 18 AWG conductor pair providing the 5V at a 250 mA drive result in approximately ¼ V drop and become the limits from the encoder's 5V rating perspective. Consideration of cable length 5V current draw is extremely important for good system design. The SERVO**STAR**'s encoder supply is 5V ± 2% with 250mA maximum draw.

When using incremental encoders, the SERVO**STAR** receives the encoder pulses, buffers them electrically and then outputs them to the encoder equivalent output port. The phase delay in the transport of these signals is extremely small.

Be sure to run all encoder cables in conduit or wire tray that is free from wires carrying or emitting electrical noises such as solenoid wires and armature cables.

Reference Variables

For additional information on any of these commands, refer to the *SERVOSTAR® SC Reference Manual*.

ENCINIT - This command triggers the encoder initialization process.

ENCINITST - This variable may be queried and returns status of the initialization process. This variable assumes one of three values:

0 = The initialization process has not started.

1 = The initialization process is in progress.

2 = The index position has been determined and the initialization process is complete. The flag is reset to zero when you manually set the index position.

ENCSTART – Selects automatic or manual "wake and shake" initialization.

HALLS - Read the states on the Hall channels.

IENCSTART - Set the "wake and shake" initialize current level.

MENCOFF - Tell the SERVO**STAR** where the marker is relative to commutation angle 0.

MENCRES - Tell the SERVO**STAR** how many lines there are on the encoder.

MENCTYPE - Tell the SERVO**STAR** what encoder signals are present.

MHINVA - Invert the active level of Hall channel A.

MHINVB - Invert the active level of Hall channel B.

MHINVC - Invert the active level of Hall channel C.

MPHASE – Allows commutation offset.

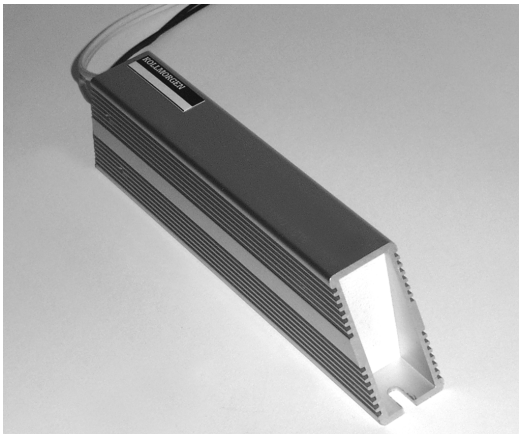
This page intentionally left blank.

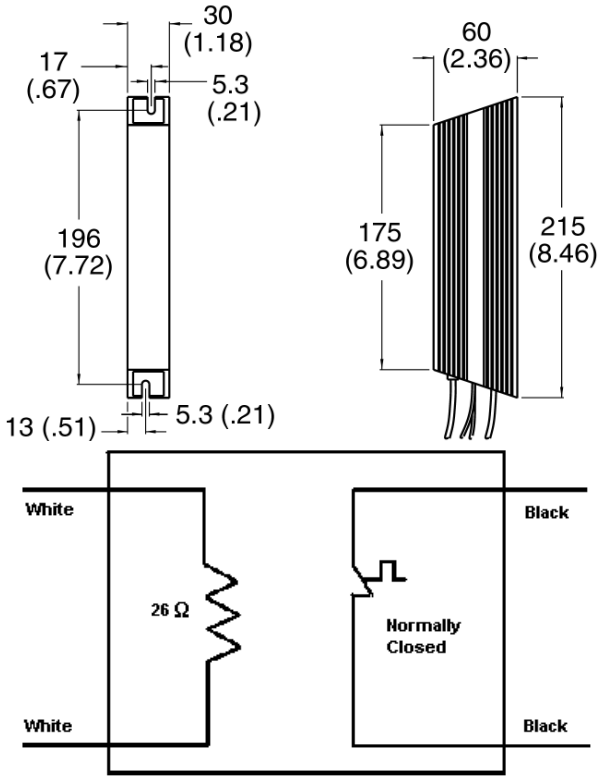
APPENDIX B

Resistive Regeneration Sizing

Shunt regeneration is required to dissipate energy that is pumped back into the DC bus during load deceleration. The amount of shunt regeneration required is a function of the sum of simultaneously decelerating loads. The loads need to be defined in terms of system inertia, maximum speed, and deceleration time. In addition, the duty cycle must be known.

Regen Resistor Example is Kollmorgen's ERH-26





The black wires are for the thermostat and the white wires are for the regen resistor on the external regen resistor (pictured above).

Energy Calculations

To determine if a system needs a regeneration resistor, use the following procedure:

EQUATION 1

Define the term E_M which is the kinetic energy of the motor/load minus the system losses.

$E_M = (1.356/2)(J_M + J_L)\omega M^2 - 3I^2M\,(R_M/2)t_d - (1.356/^2)T_F\omega_M t_d$ Joules

Where: J_M = rotor inertia (lb – ft – sec^2)

J_L = load inertia (lb – ft – sec^2)

ω_M = motor speed before decel (rad/sec) = $\frac{RPM}{9.55}$

I_M = motor current during deceleration (A_{RMS} / phase)

R_M= motor resistance (Ω L – L)

t_d = time to decel (sec)

T_F = friction torque (lb-ft)

If this energy is less than that which the BUS Module can store, then no regen resistor is needed. Thus, the condition for which no regen resistor is required is:

EQUATION 2

$E_M < (1/2)\,C\,(V^2_M - V^2_{NOM})$

Where: C = BUS Module capacitance (Farads)

V_M = max BUS voltage (V)

V_{NOM} = nominal BUS voltage (V) = V (L-L) $\sqrt{2}$

Where: all negative E_M are set equal to zero before summation (sum all non-negative E_M). This represents a worst case in which only the motors that are regenerating ($E_{MJ} > 0$) decelerate while those whose system losses exceed their regenerative energy ($E_{MJ} \leq 0$) remain idle. If Equation 2 is not satisfied, then a regeneration resistor is required.

Regeneration Calculations

The procedure for calculating regeneration requirements is twofold. Both the regen resistance value and the resistor wattage rating must be determined.

DETERMINING RESISTANCE VALUE

The maximum allowable resistance of the regen resistor is that value which will hold the BUS under its maximum value when the regen circuit is initially switched on. For an AC servo system, the maximum allowable regen resistance is given by:

EQUATION 3

$$R_{MAX} = \frac{V^2_M}{V_B I_M \sqrt{3}}$$

Where: V_M = maximum BUS voltage

V_B = motor back EMF less motor losses

EQUATION 4

$$V_B = K_B N - \sqrt{3}\, I_M (R_M / 2)$$

Where: K_B = back EMF constant (V (L-L) /KRPM)
N = motor speed prior to decel (KRPM)
I_M = deceleration current in motor (A_{RMS} / phase)
R_M = motor resistance (ΩL-L)
I_M = deceleration current in motor (A_{RMS} / phase)

DETERMINING DISSIPATED POWER

The average wattage rating of the regen resistor is a function of energy to be dissipated and the time between decelerations. This average wattage rating for a single axis system is given by:

EQUATION 5

$$P_{AV} = \frac{E_M - (1/2)C(V^2_M - V^2_{HYS})}{t_{cycle}}$$

Where: t_{cycle} = time between decels + time to decel (sec)

V_{HYS} = hysteresis point of regen circuit

When the time between decelerations becomes very large, Equation 5 become very small. In cases such as these, the average wattage is not a meaningful number. Peak wattage and the time which the resistor will see peak wattage become the main concerns. The peak wattage of the regen resistor is:

$$P_{PK} = \frac{V^2_M}{R_{REGEN}}$$

Where: R_{REGEN} = REGEN Resistance

This page intentionally left blank.

APPENDIX C

This appendix provides various examples for easy access and reference.

Subroutine Example

```
Dim Shared var_x as double
Dim Shared var_y as long
Dim Shared array_z[2][2] as double
Dim Shared rows as long
Dim Shared columns as long

Program

       REM initialize variables
       var_x = 29.3172
       var_y = 5
       rows = 2
       columns = 2

       REM specification of program
       <optional code>
              call OrdinarySubroutine(var_x, var_y)
       <more code>
              call MatrixMath(array_z, rows, columns)
       <more code>

End Program

SUB OrdinarySubroutine(param1 as double, param2 ByVal as long)

REM pass param1 by Reference, param2 by Value
       REM declare local variable is any
       REM specification of subroutine
       <optional code using param1 and param2>

END SUB

SUB MatrixMath(matrix as double, rows ByVal as long, columns ByVal as long)

REM pass Matrix by Reference, rows by Value, columns by Value)
       REM declare local variables if any
       REM specification of subroutine
       <optional code using matrix, rows, and columns>

END SUB
```

ContinueTask Example

If you want a task to resume after an error has occurred, you should use the ContinueTask command.

```
OnError

       Catch 8001
       Print "division by zero"
       ContinueTask Task1.prg

End OnError
```

In this example, the name of the task containing this OnError block is TASK1.PRG. If you do not add the CONTINUETASK command to the end of the OnError block, the program remains idled after the OnError block is executed.

OnError Example

Below is an example OnError rountine that catches when the remote enable input on connector C3 is removed.

```
OnError
Catch 3058
Print "Enable signal removed - Task2 stopped"
 'Print to Message Log
while REMOTE = 0                        'Wait for HW en to closed
       Sleep 10
End While

ContinueTask Task2.prg                  'Continue task
Print "Task2 restarted"                 'Print to Message Log

Catch else
sleep 2

End OnError
```

Axis Setup Subroutine Example

```
rem ----------------------------------------------------------------
rem TASK....: IndexMove.prg
rem AUTHOR..: John Doe
rem TIME....: August 15, 2000
rem PURPOSE.: Main Program for Simple Move Project
rem ----------------------------------------------------------------
rem Declare Variables Here
Dim Shared sample_int as Long
Dim Shared sample_real as Double
rem ----------------------------------------------------------------
rem                   Main Task Begins Here
rem ----------------------------------------------------------------
Program
       Attach                           'Attach to axis
       Call AxisSetup                   'Set up all axes
Rem          Ensure Drive is in Conmode =2 and enable drive
       If  en<>1 then
       Sys.Conmode=2                    'Set Conmode = 2
       En = 1                           'Enable drive
       EndIf
       Sleep 1000                       'Sleep one second
              For sample_int = 1 To 10   'Loop 10 times
rem    Move axis 65536 counts in positive direction
       Move 65536
       Sleep 2000                     'Sleep two seconds
rem    Move axis 65536 counts in the negative direction
       Move -65536
       Sleep 2000                  'Sleep two seconds
       Next sample_int
       Print "Sample program completed."
       En = off                      'Disable system
       StartType = GeneratorCompleted
rem    Auto generated sample program end here
       Detach                      'Detach from axis
End Program
rem ----------------------------------------------------------------
rem    Axis Set-up Routine Automatically Generated by the Project Wizard
rem ----------------------------------------------------------------
Sub AxisSetup
       En = off                    'Make sure axis is disabled
       Acceleration  = Amax        'Set Acceleration to the Maximum
       Deceleration  = Dmax        'Set Deceleration to the Maximum
       VCruise  = 50               'Set The Cruise Velocity to 50 RPM
       PEMax = 1000                'Set Maximum Position Error to 1000 Counts
       Disp = 0                    'Set Displacement to 0
       StartType = Immed           'Set Start Type to GCOM
       Absolute = 0                'Set to Incremental Position Mode

End Sub
```

The program has these sections.

- A short task-variable declaration
- An example program with:
  - Intro section that calls the axis setup soubroutine and then enables the drive
  - A section that generates simple motion
- An axis setup subroutine.

The variable declaration section provides examples of variable declarations for a task.

The motion program cycles the axis 10 times. Most of this main program is commented out because this code enables the SERVO**STAR** and generates motion commands. You must remove the single-quote comments markers before the program will run. You must ensure your machine is safe to operate before executing this program. This includes tuning your axis properly to assure stable motion control. Refer to the *SERVO**STAR**® SC Installation Manual* for more information.

The final section of the auto setup program is the axis setup subroutine. This subroutine loads axis properties for limits. You do not need to understand the program in detail at this point, but you should familiarize yourself with the structure.

Rotary Mode Example

Consider an example where the axis drives a rotary table through a 5:3 gearbox:

Desired units:	Degrees
Desired repeat:	360
Gearbox:	5:3
Feedback	2000 line encoder

For this example, first set position units to degrees:

1 degree	=	1/360 revolution of table
	=	(5/3) * (1/360) revolution of the motor
	=	2000 * (5/3) * (1/360) lines
	=	4 * 2000 * (5/3) * (1/360) counts
	=	40000/1080

After setting these units, you must enable the rotary mode and set the rollover to 360 as follows:

```
PositionRollover = 360 * 40000 / 1080
PositionRolloverEnable = Rotary                'Enable Rotary Motion
```

In this case, the feedback position is always between 0 and 360 * 40000 / 1080. You can still command long moves, say 10,000 degrees. However, when the motor comes to rest, the feedback is read as less than 360.

POSITIONROLLOVERENABLE defaults to “Linear”.

Motion Examples

This section provides a few examples that apply the motion control techniques to common machine functions.

Homing Example

Here is a sample HOME program that is set up as a subroutine.

You can place this code in the main program if desired.

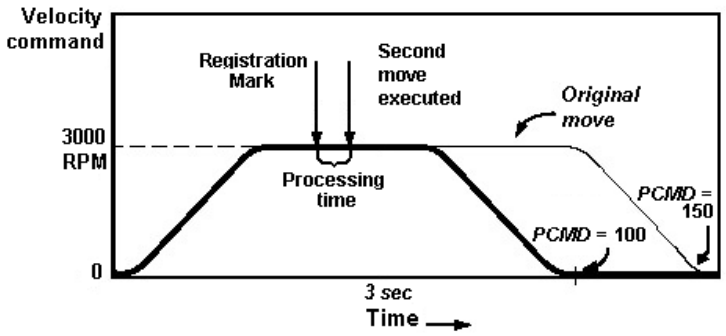
```
Sub Home1

In3Mode = 10                'Sets In3 on Connector C3 pin 11 as home switch input
HomeType = 0                'Home to switch plus marker
HomeReturn = 1              'Move distance specified by Homeoffset after reaching marker
HomeVelocity = 10           'Home velocity = 10 rpm
HomePolarity = 1            'Homing active on rising edge
HomeOffset = 20000          'Offset home postion by 20000 counts from marker
HomeDistance = 15000        'Set postion (PFB) when home complete to 15000
HomeDistancemax = 500000    'Set maximun on homing distance to 500000 counts
Home                        'Start Homing Move
While Homestatus <> 3       'Check home status and wait for home completion
   Sleep 2
End While

Print " Home Complete"      'Print To Mesage log in BASIC Moves or MotionSuite

End Sub
```

Registration Example

The following is a typical registration example:

```
Sub RegistrationExample

REM Prepare SERVOSTAR SC to Capture
Attach                      'Make sure axis is attached to task
In1Mode = 16                'Set Input IN1 to Capture Function
CapturePolarity = 1                 'Capture Position on Rising Edge of input
Capture = 1                         'Start Capture Procedure
CaptureArm = 0                      'Reset Capture Input
StartCapture:
       CaptureArm = 1                       'Arm Capture Input
REM Move 1000000 Counts at a velocity of 3000 RPM
       Move 1000000 VCruise = 300 Absolute = 0
REM Wait for Capture Input
       While CaptureStatus = 0
              Sleep 1
       End While
REM Move to 100000 counts past Capture Position
       Move CapturePosition + 10000  StartType = Immed Absolute = 1
Rem Wait for motion to stop
       While IsMoving
              Sleep 1
       End While
CaptureArm = 0
Goto StartCapture
End Sub
```

This example generates the profile shown in the graph below.



As shown in the graph, the move starts and the registration mark is detected about 40% of the way to the end. After a small amount of processing time, the second move is loaded over the current move. The axis then comes to rest a fixed offset after the registration mark.

As in homing, the more accurately the position is determined, the more accurate the cut is with respect to the registration mark. If the process is feeding material at 10 feet-per-second, and the SERVO**STAR** SC capture accuracy is ±3 microseconds, the mark is determined to an accuracy of 10 (feet per sec) * 3-uSec = 3.225e-5 ft or about 0.0004 in.

As you can see in this picture, the original line was set to go much farther than the actual move that resulted. This technique is commonly used because you can monitor the end position. If the move is completed before the registration mark is detected, it indicates that the mark was missed.

APPENDIX D: DEVICE NET

Functionality Chart

DeviceNet™ Device Type	ODVA Requirements Position Controller
Explicit Peer-to-Peer Messaging	N
I/O Peer-to-Peer Messaging	N
Baud Rate	500 kB
Master/Scanner	N
I/O Slave Messaging	
Bit Strobe	N
Polling	Y
Cyclic	N
Change-of-State (COS)	N

Object Model

These are objects supported by the unit in the Position Controller Device.

Object: Identity

Class Code 0x01
Instance # 1
Description This object provides identification of any general information about the device.

Object: Messsage Router

Class Code 0x02
Instance # 1
Description This object provides a messaging connection point through which a client may address a service to any object class or instance residing in the physical device.

Object: DeviceNet

Class Code 0x03
Instance # 1
Description This object provides the configuration and status of a DeviceNet port.

Object: Assembly

Class Code 0x04
Instance # 1
Description This object binds attributes of multiple objects that allows data to or from each object to be sent or received over a single connection. Assembly objects can be used to bind input or output data. An input produces data on the network and an output uses data from the network.

Object: Assembly

Class Code 0x04
Instance # 2
Description This object stores I/O output message data.

Object: Explicit Connection

Class Code 0x05
Instance # 1
Description This object manages the explicit messages.

Object: I/O Connection

Class Code 0x05
Instance # 2
Description This object manages the I/O messages.

Object: Position Controller Supervisor

Class Code 0x24
Instance # 1
Description The position controller supervisor handles errors for the position controller as well as home inputs.

Object: Position Controller

Class Code 0x25
Instance # 1
Description The position controller object performs the control output velocity profiling and handles input and output to and from the motor drive unit, limt switches, etc.

Object: Block Sequencer

Class Code 0x26
Instance # 1
Description This object handles the execution of command blocks or command block chains.

Object: Command Block

Class Code 0x27
Instance # 1 to 255
Description Each instance of the command block object defines a specific command. These blocks can be linked to other blocks to form a command block chain.

Position Controller Data Types

Supported Services

The only services supported by the Kollmorgen DeviceNet Block Command Object, Block Sequence Object, Position Controllers, Position Controller Supervisor Object and Position Controller Object are:

Get_Single_Attribute (service code 14 0x0E) and
Set_Single_Attribute (service code 16 0x10).

Also note that the **Axis Instance** is always 1.

For additional information, we recommend that you review this entire document.

Data Types

The table below describes the data type, number of bits, minimum and maximum Range.

Data Type	Number of Bits	Minimum Value	Maximum Value
Boolean	1	0 (False)	1 (True)
Short Integer	8	-128	127
Unsigned Short Integer	8	0	255
Integer	16	-32768	32767
Unsigned Integer	16	0	65535
Double Integer	32	-2^{31}	$2^{31} - 1$
Unsigned Double Integer	32	0	$2^{32} - 1$

Position Controller Supervisor Object Class (ID=36)

Error Codes

The drive returns one of the following codes when an error is generated while communicating via Explicit Messaging.

Action	Error	Error Code
Set	Attribute Not Settable	0x0E
Set or Get	Attribute Not Supported	0x14
Set or Get	Service Not Supported	0x08
Set or Get	Class Not Supported	0x16
Set	Value is Out of Range	0x09
Set or Get	Not Enough Data	0x13
Set or Get	Too Much Data	0x15

Supervisor Attributes

These are attributes supported by the unit in the Position Controller Supervisor Class.

Attribute ID 1: Number of Attributes

Access Rule Get
Data Type Unsigned Short Integer
Description The total number of attributes supported by the unit in the Position Controller Supervisor Class.
Range This is always 23.
Default 23

Attribute ID 2: Attribute List

Access Rule Get
Data Type Array of Unsigned Short Integer
Description Returns an array with a list of the attributes supported by this unit in the Position Controller Supervisor Class. The length of this list is specified in Number of Attributes.
Range Array size is defined by Attribute 1.
Default 1-3, 5-7, 11, 12, 16, 17, 19-28, 253-255.

Attribute ID 3: Axis Number

Access Rule Get
Data Type Unsigned Short Integer
Description Returns the axis number which is the same as the instance for this object.
Range This is always 1.
Default 1

Attribute ID 5: General Fault

Access Rule Get
Data Type Boolean
Description When active, this indicates that a drive-related failure has occurred (Short Circuit, Over-Voltage, etc.). It is not related to the FAULT input. It is reset when the fault condition is removed. For a description of drive fault, see Attribute 251 in Class 37.
Range 1 = Fault condition exists
0 = No fault exists
Default None

Attribute ID 6: Input Command Assembly Type

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute specifies which Input Command Assembly Type is used during polled I/O commands. See the Polled I/O Command Assemblies section for additional details.
Range	Drive Dependent (See the Polled I/O Command Assemblies section for more information.)
Default	0x00

Attribute ID 7: Response Assembly Type

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	Sets the response message that is returned to the controlling device.
Range	Valid Message Type Codes are:0x00, 0x01, 0x02, 0x03, and 0x1E
Default	0x00

Attribute ID 8: Fault Input

Access Rule	Get
Data Type	Boolean
Description	This attribute reflects the state of the Hardware Fault Input.
Range	0 = Fault Input Inactive 1 = Fault Input Active
Default	None

Attribute ID 11: Home Active Level

Access Rule	Get Set
Data Type	Boolean
Description	This attribute defines the polarity of the homing switch.
Range	0 = Home active when homing switch is "Low" 1 = Home active when homing switch is "High"
Default	None

Attribute ID 12: Home Arm

Access Rule	Get Set
Data Type	Boolean
Description	This attribute arms the Home Input.
Range	0 = Getting it indicates the trigger has occurred 1 = Arms the home input
Default	None

Attribute ID 16: Home Input Level

Access Rule	Get
Data Type	Boolean
Description	This attribute reflects the raw state of the Hardware Home Input.
Range	0 = Home Input "Low" 1 = Home Input "High"
Default	None

Attribute ID 17: Home Position

Access Rule	Get
Data Type	Double Integer
Description	This attribute reflects the position at the time the home input is triggered Home Position always equals to 0.
Range	In the end of homing procedure controller sets Home Position to 0.
Default	0

Attribute ID 19: Registration Action

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute defines what happens when the registration input is triggered.
Range	0 – Command Output Generator off (drive becomes disable) 1 – Hard Stop 2 – Smooth Stop 3 – No action 4 – Go to Reg. position offset 5 – Go to Reg. position absolute
Default	0

Attribute ID 20: Registration Active Level

Access Rule	Get Set
Data Type	Boolean
Description	This attribute defines the polarity of the registration switch.
Range	0 = Registration active when registration switch is "Low" 1 = Registration active when registration switch is "High"
Default	1

Attribute ID 21: Registration Arm

Access Rule	Get Set
Data Type	Boolean
Description	This attribute arms the Registration input.
Range	0 = Getting it indicates the registration trigger has occurred 1 = To arm registration input
Default	0

Attribute ID 22: Registration Input Level

Access Rule	Get
Data Type	Boolean
Description	This attribute reflects the actual level of the Registration input.
Range	0 = Registration input is "Low" 1 = Registration input is "High"
Default	No

Attribute ID 23: Registration Offset

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute defines a position value used as the offset or absolute position, dependent on the registration action code.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	0

Attribute ID 24: Registration Position

Access Rule	Get
Data Type	Double Integer
Description	This attribute reflects the position at the time the Registration Input is triggered.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	0

Attribute ID 25: Follow Enable

Access Rule	Get Set
Data Type	Boolean
Description	This attribute enables following of the Follow Axis.
Range	0 = following disabled 1 = following enabled
Default	0

Attribute ID 26: Follow Axis

Access Rule	Get Set
Data Type	Integer
Description	This attribute specifies the Axis to follow. *The SERVO**STAR** SC only has one axis so this value CANNOT be more than 1.*
Range	0 = No following 1 = Following
Default	0

Attribute ID 27: Follow Divisor

Access Rule	Get Set
Data Type	Double Integer
Description	Divide the Follow Axis position by this Follow Divisor to calculate the Command Position.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	1

Attribute ID 28: Follow Multiplier

Access Rule	Get Set
Data Type	Double Integer
Description	Multiply the Follow Axis position by the Follow Multiplier value to calculate the Command Position.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	1

Attribute ID 253: Input for Registration Mode

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute defines the input for Registration Mode. *This is NOT a standard DeviceNet attribute.* ***Only one input can be used for Registration***.
Range	0 = no input is used for registration 1 = input 1 is used for registration 2 = input 2 is used for registration 3 = input 3 is used for registration
Default	0

Attribute ID 254: Input for Home Mode

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute defines the input for Home Mode. *This is NOT a standard DeviceNet attribute.* ***Only one input can be used for Homing***.
Range	0 = no input is used for homing 1 = input 1 is used for homing 2 = input 2 is used for homing 3 = input 3 is used for homing
Default	3

Attribute ID 255: Follow Master Source

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute defines the source of an axis when it is in a Slave mode. The slave axis follows the command generated by that source.*This is NOT a standard DeviceNet attribute.*
Range	0 = POSITIONEXTERNAL (External Position from any axes) 1 = POSITIONFEEDBACK (Position Feedback from any axes) 2 = POSITIONCOMMAND (Position command of any axis)
Default	0

This page intentionally left blank.

Position Controller Object Class (ID=37)

Error Codes

The drive returns one of the following error codes when an error is generated while communicating via Explicit Messaging.

Action	Error	Error Code
Set	Attribute Not Settable	0x0E
Set or Get	Attribute Not Supported	0x14
Set or Get	Service Not Supported	0x08
Set or Get	Class Not Supported	0x16
Set	Value is Out of Range	0x09
Set or Get	Not Enough Data	0x13
Set or Get	Too Much Data	0x15

Attribute 1: Number of Attributes

Access Rule	Get
Data Type	Unsigned Short Integer
Description	The total number of attributes supported by the unit in the Position Controller Class.
Range	This is always 47.
Default	47

Attribute 2: Attribute List

Access Rule	Get
Data Type	Array of Unsigned Short Integer
Description	Returns an array with a list of the attributes supported by this unit in the Position Controller Class. The length of this list is specified in Number of Attributes.
Range	Array size is defined by Attribute 1.
Default	1-3, 6-17, 20, 21, 23, 24, 30-33, 35-38, 40, 41, 44-58, 251-254

Attribute 3: Mode

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute defines the operation mode.
Range	0 = Position Mode 1 = Velocity Mode
Default	0

Attribute 6: Target Position

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies the target position in counts.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	0

Attribute 7: Target Velocity

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies the target velocity in counts per second.
Range	Set to a positive number
Default	According to setup

Attribute 8: Acceleration

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies the acceleration for positioning, continuous velocity and homing in counts per second^2.
Range	Set to a positive number
Default	According to setup

Attribute 9: Deceleration

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies the deceleration for positioning, continuous velocity and homing in counts per second^2.
Range	Set to a positive number
Default	According to setup

Attribute 10: Incremental Position Flag

Access Rule	Get Set
Data Type	Boolean
Description	This bit is used to define the position value as either absolute or incremental.
Range	0 = Absolute Position 1 = Incremental Position
Default	1

Attribute 11: Trajectory Start/Complete

Access Rule	Get Set
Data Type	Boolean
Description	Sets a start a trajectory move. Reads cleared when a profile move is complete.
Range	0 = Move Complete 1 = Start Trajectory (In Motion)
Default	0

Attribute 12: On Target Position

Access Rule	Get
Data Type	Boolean
Description	This flag, when set, indicates that the motor is within the deadband distance to the target.
Range	0 = Not On Target Position 1 = In Position
Default	1

Attribute 13: Actual Position

Access Rule	Get Set
Data Type	Double Integer
Description	The absolute position value equals the real position in counts. This is set to re-define the actual position.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	0

Attribute 14: Actual Velocity

Access Rule	Get
Data Type	Double Integer
Description	This attribute specifies the actual velocity in counts per second.
Range	Positive read value
Default	0

Attribute 15: Commanded Position

Access Rule	Get
Data Type	Double Integer
Description	This value equals the instantaneous calculated position.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	0

Attribute 16: Commanded Velocity

Access Rule	Get
Data Type	Double Integer
Description	This value equals the instantaneous calculated velocity in profile units per second.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	0

Attribute 17: Enable

Access Rule	Get Set
Data Type	Boolean
Description	This flag is used to control the enable output. Clearing this bit sets the enable output inactive and the currently executing motion profile is aborted.
Range	0 = Disable 1 = Enable
Default	0

Attribute 20: Smooth Stop

Access Rule	Get Set
Data Type	Boolean
Description	This bit is used to bring the motor to a controlled stop at the currently implemented deceleration rate. Deceleration in the SERVO**STAR** SC can only stop in maximal deceleration (Hard Stop attribute).
Range	0 = No Action 1 = Perform Smooth Stop
Default	0

Attribute 21: Hard Stop

Access Rule	Get Set
Data Type	Boolean
Description	This bit is used to bring the motor to an immediate stop.
Range	0 = No Action 1 = Perform Hard Stop
Default	0

Attribute 23: Direction

Access Rule	Get Set
Data Type	Boolean
Description	This bit is used to control the direction of the motor in profiled velocity mode.
Range	0 = Negative Direction 1 = Positive Direction
Default	1

Attribute 24: Reference Direction

Access Rule	Get Set
Data Type	Boolean
Description	Defines positive direction (when viewed from the motor shaft side).
Range	0 = Positive Clockwise Motion 1 = Positive Counter-Clockwise Motion
Default	0

Attribute 30: Kp

Access Rule	Get Set
Data Type	Unsigned Integer
Description	This attribute defines the proportional gain for the PID position loop controller.
Range	0 - 7000
Default	Calculated

Attribute 31: Ki

Access Rule	Get Set
Data Type	Unsigned Integer
Description	This attribute defines the position loop integral gain for the Proportional-Integral-Derivative (PID), position loop controller.
Range	0 - 10000
Default	0

Attribute 32: Kd

Access Rule	Get Set
Data Type	Integer
Description	This attribute defines the position loop derivative gain for the Proportional-Integral-Derivative (PID) position loop controller.
Range	0 - 32767
Default	0

Attribute 35: Velocity Feed Forward

Access Rule	Get Set
Data Type	Unsigned Integer
Description	This attribute defines velocity feed forward gain value.
Range	0 - 2000
Default	0

Attribute 36: Acceleration Feed Forward

Access Rule	Get Set
Data Type	Unsigned Integer
Description	This attribute defines acceleration feed forward gain value.
Range	0 - 2000
Default	0

Attribute 37: Sample Rate

Access Rule	Get
Data Type	Integer
Description	Constant update sample rate in µSeconds.
Range	500
Default	Always 500 µS.

Attribute 38: Position Deadband

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	Set to prevent axis hunting within the desired window.
Range	0 to 255 (0.0 to 255 revolutions)
Default	0

Attribute 40: Feedback Resolution

Access Rule	Get Set (only for encoder and sign encoder)
Data Type	Double Integer
Description	This attribute defines the motor's rotary or linear feedback resolution.
Range	100 - 1000000
Default	Encoder: motor data Resolver: 65536

Attribute 41: Motor Resolution

Access Rule	Get Set (only for encoder and sign encoder)
Data Type	Double Integer
Description	This attribute defines the motor resolution in motor steps and is the same as attribute Feedback Resolution (attribute 40).
Range	100 - 1000000
Default	Encoder: motor data Resolver: 65536

Attribute 44: Max Static Following Error

Access Rule	Get Set
Data Type	Double Integer
Description	Maximum allowable following error when the motor stopped and holding position. This attribute is the same as attribute Max Dynamic Following Error (attribute 45).
Range	0 – Maximum Double Integer
Default	1000000

Attribute 45: Max Dynamic Following Error

Access Rule	Get Set
Data Type	Double Integer
Description	Maximum allowable following error when the motor is in motion. ***This attribute is the same as Max Static Following Error (attribute 44).***
Range	0 – Maximum Double Integer
Default	1000000

Attribute 46: Following Error Action

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	Following error action code.
Range	0 = Command Output Generator Off (drive becomes disable) 1 = Hard Stop 2 = Smooth Stop 3 = No action
Default	0

Attribute 47: Following Error Fault

Access Rule	Get Set
Data Type	Boolean
Description	Following error occurrence flag. Setting this attribute to 0 clears an existing fault. This bit can be cleared directly or by reprogramming the Following Error Action attribute.
Range	0 = When another move is attempted 1 = When a following error occurs
Default	0

Attribute 48: Actual Following Error

Access Rule	Get
Data Type	Double Integer
Description	This attribute returns the position following error, which is the difference between position command and the position feedback.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	0

Attribute 49: Hard Limit Action

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	Defines the response to a hard limit.
Range	1 = Hard Stop
Default	1

Attribute 50: CW Limit Input Active

Access Rule	Get
Data Type	Boolean
Description	This attribute is set when CW hard limit is active. Motion is not allowed in the positive direction when active.
Range	0 = Not Active 1 = Active
Default	0

Attribute 51: CCW Limit Input Active

Access Rule	Get
Data Type	Boolean
Description	Set when CCW hard limit is active. Motion is not allowed in the negative direction when active.
Range	0 = Not Active 1 = Active
Default	0

Attribute 52: Soft Limit Enable

Access Rule	Get Set
Data Type	Boolean
Description	Enables the soft limits. When set, a move that exceeds the defined limits results in a motor stop.
Range	0 = Disable 1 = Enable
Default	0

Attribute 53: Soft Limit Action

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	Defines the response to a positive or negative software limit.
Range	0 = Disable 1 = Hard Stop (infinite decel) 2 = Smooth Stop (decel to stop)
Default	2

Attribute 54: Positive Soft Limit Position

Access Rule	Get Set
Data Type	Double Integer
Description	Sets the positive soft limit position in counts.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	$2^{31}-1$

Attribute 55: Negative Soft Limit Position

Access Rule	Get Set
Data Type	Double Integer
Description	Sets the negative soft limit position in counts.
Range	$-2^{31}+1$ to $2^{31}-1$
Default	$-2^{31}+1$

Attribute 56: Positive Soft Limit Triggered

Access Rule	Get
Data Type	Boolean
Description	Monitors the soft positive position limit.
Range	0 = Positive Soft Limit Not Reached 1 = Positive Soft Limit Reached
Default	0

Attribute 57: Negative Soft Limit Triggered

Access Rule	Get
Data Type	Boolean
Description	Monitors the soft negative position limit.
Range	0 = Negative Soft Limit Not Reached 1 = Negative Soft Limit Reached
Default	0

Attribute 58: Load Data Complete

Access Rule	Get
Data Type	Boolean
Description	This attribute indicates that valid data for a valid I/O command message type has been loaded into the position controller device.
Range	0 = Load Data Complete 1 = Position controller device will attempt to load the data contained in command message data bytes.
Default	0

Attribute 251: Drive Status

Access Rule	Get
Data Type	Double Integer
Description	Returns drive status. Each bit of this attribute represents a different error.
Range	Bit 0: Overload fault Bit 1:Amplifier over temperature fault Bit 2:Motor over temperature fault Bit 3:Reserved: Cooling system fault (set to 0) Bit 4:Control voltage fault (analog supply failure) Bit 5:Feedback loss fault Bit 6:Commutation fault Bit 7:Over current fault Bit 8:Over voltage fault Bit 9:Under voltage fault Bit 10:Reserved: Power supply phase fault (set to 0) Bit 11:Excessive position deviation Bit 12:Communication interface fault Bit 13:Software limit switch fault Bit 14:Reserved (set to 0) Bit 15:Non-volatile data memory fault Bit 16:Non-volatile data memory checksum fault Bit 17:Reserved Bit 18: Reserved Bit 19: Reserved Bit 20: Reserved Bit 21: Reserved Bit 22: Reserved Bit 23:Invalid drive/motor configuration Bit 24:Motor over speed fault Bit 25:Reserved Bit 26:Reserved Bit 27:Positioner synchronization error Bit 28:Positioner synchronization error Bit 29:External communication fault Bit 30:Internal firmware fault Bit 31:Positioner fault. - Check SYS.MOTION flag
Default	0

Attribute 252: Homing Type

Access Rule	Get Set
Data Type	Unsigned Single Integer
Description	Homing type.
Range	0 = Home flag and index or Z phase of the encoder define home 1 = Home flag defines home 3 = Index or Z phase of the encoder defines home
Default	0

Attribute ID 253: Home Direction

Access Rule	Get Set
Data Type	Boolean
Description	This flag defines the initial direction of the move for homing.
Range	0 = Negative Direction 1 = Positive Direction
Default	1

Attribute ID 254: Home Velocity

Access Rule	Get Set
Data Type	Double Integer
Description	The fast velocity used during homing in units of steps per second (1LSB = 1 step/sec).
Range	Positive number
Default	

Block Sequencer Object Class (ID=38)

This object handles the execution of Command Blocks or Command Block chains.

Attribute 1: Block

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This value defines the starting Command Block instance number to execute.
Range	1 to 255
Default	N/A

Attribute 2: Block Execute

Access Rule	Get Set
Data Type	Boolean
Description	Executes the starting command block defined by Attribute 1.
Range	0 = Clear or Complete 1 = Block Executing
Default	0

Attribute 3: Current Block

Access Rule	Get
Data Type	Unsigned Short Integer
Description	This attribute returns the command block instance number of the currently-executing block.
Range	1 to 255
Default	N/A

Attribute 4: Block Fault

Access Rule	Get
Data Type	Boolean
Description	Set when a block error occurs. When a block fault error occurs, block execution stops. This bit is reset when the block fault code (5) is read.
Range	0 = No Block Faults 1 = Block Fault Occurred
Default	0

Attribute 5: Block Fault Code

Access Rule	Get
Data Type	Boolean
Description	This attribute defines the specific block fault.
Range	0 = No Fault 1 = Invalid or Empty Block 2 = Command Time-out (Wait Equals) 3 = Execution Fault
Default	0

Command Block Object Class (ID=39)

Each instance of the Command Block object defines a specific command. These blocks can be linked to other blocks to form a command block chain. Looping and branching commands are supported. See the individual commands for additional attribute information.

Command 01 – Modify Attribute

This command is used to modify an Attribute's value.

Attribute 1: Block Command

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute specifies the command to be performed.
Range N/A
Default 0x01 = Command 01

Attribute 2: Block Link #

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range 0 to 255 ; 0 – means this is the last block in linked list
Default 0

Attribute 3: Target Class

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the target class number to be sequenced.
Range
Default

Attribute 4: Target Instance

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the instance number of the target class to be sequenced.
Range
Default

Attribute 5: Attribute #

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the Position Controller Class Attribute Number. The Position Controller Class Attribute Number must be settable.
Range
Default

Attribute 6: Attribute Data

Access Rule	Get Set
Data Type	Double Integer
Description	This is the new attribute data.
Range	
Default	

Command 02– Wait Equals Command

This command is used to stop execution of a linked chain of commands until an attribute becomes valid.

Attribute 1: Block Command

Access Rule Get
Set
Data Type Integer
Description This attribute specifies the command to be performed.
Range 0x02 = Command 02
Default

Attribute 2: Block Link #

Access Rule Get
Set
Data Type Integer
Description This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range
Default

Attribute 3: Target Class

Access Rule Get
Set
Data Type Integer
Description This attribute defines the target class number to be sequenced.
Range
Default

Attribute 4: Target Instance

Access Rule Get
Set
Data Type Integer
Description This attribute defines the instance number of the target class to be sequenced.
Range
Default

Attribute 5: Attribute #

Access Rule Get
Set
Data Type Integer
Description This attribute defines the Position Controller Class Attribute Number. The Position Controller Class Attribute Number must be settable.
Range
Default

Attribute 6: Compare Time-Out Value

Access Rule Get
Set
Data Type Double Integer
Description This attribute compares time-out value in milliseconds. If the time-out compare doesn't occur, a fault is generated and motion stops.
Range 0 to 2^{31}-1
Default 0 = No Time-out

Attribute 7: Compare Data

Access Rule Get
Set
Data Type Dependent on Attribute #
Description This attribute compares the data for the end of command. If Attribute 6 is equal to the compare data, the block is complete and the next linked block is executed.
Range
Default

Command 03– Conditional Link Greater Than Command

This command is used for conditional linking or branching in a linked chain of commands.

Attribute 1: Block Command

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute specifies the command to be performed.
Range 0x03 = Command 03
Default

Attribute 2: Block Link #

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range
Default

Attribute 3: Target Class

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the target class number to be sequenced.
Range
Default

Attribute 4: Target Instance

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the instance number of the target class to be sequenced.
Range
Default

Attribute 5: Attribute #

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the Position Controller Class Attribute Number. The Position Controller Class Attribute Number must be settable.
Range
Default

Attribute 6: Compare Link #

Access Rule Get
Set
Data Type Double Integer
Description This is the conditional link number or the alternate link block if this attribute is greater than the compare data.
Range
Default

Attribute 7: Compare Data

Access Rule Get
Set
Data Type Dependent on Attribute #
Description This attribute compares the data for the conditional link. If Attribute 6 is greater than the compare data, the normal link (Attribute 2) is ignored and the next block executed is the compare link block.
Range
Default

Command 04– Conditional Link Less Than Command

This command is used for conditional linking or branching in a linked chain of commands.

Attribute 1: Block Command

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute specifies the command to be performed.
Range 0x04 = Command 04
Default N/A

Attribute 2: Block Link #

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range Depends on command
Default

Attribute 3: Target Class

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the target class number to be sequenced.
Range
Default

Attribute 4: Target Instance

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the instance number of the target class to be sequenced.
Range
Default

Attribute 5: Attribute #

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the Position Controller Class Attribute Number. The Position Controller Class Attribute Number must be settable.
Range
Default

Attribute 6: Compare Link #

Access Rule Get
Set
Data Type Double Integer
Description This is the conditional link number or the alternate link block if this attribute is less than the compare data.
Range
Default

Attribute 7: Compare Data

Access Rule Get
Set
Data Type Dependent on Attribute #
Description This attribute compares the data for the conditional link. If Attribute 6 is less than the compare data, the normal link (Attribute 2) is ignored and the next block executed is the compare link block.
Range
Default

Command 06– Delay Command

This command is used to delay a linked chain of commands.

Attribute 1: Block Command

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute specifies the command to be performed.
Range	0x06 = Command 06
Default	N/A

Attribute 2: Block Link #

Access Rule	Get Set
Data Type	Unsigned Short Integer
Description	This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range	Depends on command
Default	

Attribute 3: Delay

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute sets the delay in milliseconds.
Range	1 to 2^{31} -1
Default	

Command 07– Trajectory Command

This command is used to initiate a move.

Attribute 1: Block Command

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute specifies the command to be performed.
Range 0x07 = Command 07
Default N/A

Attribute 2: Block Link #

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range
Default

Attribute 3: Target Position

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the target position in units of steps.
Range
Default

Attribute 4: Target Velocity

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the target velocity in units of steps per second.
Range Positive number
Default

Attribute 5: Incremental

Access Rule	Get Set
Data Type	Boolean
Description	This flag defines if the motion is incremental or absolute.
Range	0 = Absolute Position 1 = Incremental Position
Default	0

Command 08– Trajectory Command and Wait

This command is used to initiate a move and wait for completion.

Attribute 1: Block Command

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute specifies the command to be performed.
Range 0x08 = Command 08
Default N/A

Attribute 2: Block Link #

Access Rule Get
Set
Data Type Unsigned Short Integer
Description This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range
Default

Attribute 3: Target Position

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the target profile position in position units.
Range
Default

Attribute 4: Target Velocity

Access Rule Get
Set
Data Type Double Integer
Description This attribute defines the target profile velocity in profile units per second.
Range
Default

Attribute 5: Incremental

Access Rule	Get Set
Data Type	Boolean
Description	This flag defines if the motion is incremental or absolute.
Range	0 = Absolute Position 1 = Incremental Position
Default	0

Command 09– Velocity Change Command

This command is used to initiate change in velocity a move and wait for completion.

Attribute 1: Block Command

Access Rule	Get Set
Data Type	Integer
Description	This attribute specifies the command to be performed.
Range	0x09 = Command 09
Default	

Attribute 2: Block Link #

Access Rule	Get Set
Data Type	Integer
Description	This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range	
Default	

Attribute 3: Target Velocity

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies the target velocity in counts per second.
Range	Set to a positive number
Default	According to setup

Command 10– Go To Home Command

This command is used to perform a move to the captured Home position.

Attribute 1: Block Command

Access Rule	Get Set
Data Type	Integer
Description	This attribute specifies the command to be performed.
Range	0x0A = Command 10
Default	

Attribute 2: Block Link #

Access Rule	Get Set
Data Type	Integer
Description	This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range	
Default	

Attribute 3: Home Offset

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies home position offset. (The offset plus the captured Home position equals the absolute target position).
Range	-2^{31} to 2^{31}
Default	

Attribute 4: Target Velocity

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies the target velocity in counts per second.
Range	Set to a positive number
Default	According to setup

Command 12– Go To Registration Command

This command is used to perform a move to the captured registration position.

Attribute 1: Block Command

Access Rule	Get Set
Data Type	Integer
Description	This attribute specifies the command to be performed.
Range	0x0C = Command 12
Default	

Attribute 2: Block Link #

Access Rule	Get Set
Data Type	Integer
Description	This attribute provides a link to the next block instance to execute. When this block is complete, the link block is executed.
Range	
Default	

Attribute 3: Registration Offset

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies index position offset. (The offset plus the captured index position equals the absolute target position).
Range	-2^{31} to 2^{31}
Default	

Attribute 4: Velocity

Access Rule	Get Set
Data Type	Double Integer
Description	This attribute specifies the target velocity in counts per second.
Range	Set to a positive number
Default	According to setup

Polled I/O Command Assemblies

Polled I/O Command Assemblies are a method of communicating to devices a group of specific commands. This method of communication is preferred, as it is faster than explicit messaging. In this section, the format for each Command Assembly is defined and examples of each are provided.

NOTE

All eight bytes of data are ignored unless a valid command assembly type is specified in byte 2 (valid command assembly types are 1 through 5).

NOTE

A valid Response Command Assembly, (decimal; 1 to 8) is required before setting any other attribute. The controllers do not respond if the Response Command Assembly is not valid.

NOTE

Data outside the range of the attribute is ignored and the Invalid Poll Data bit of the Response Assembly is set. This applies to all Command Assemblies, except Assembly 1.

Command Assembly 1: Target Position

Description This command assembly is used to start a trajectory (position mode only) of the specified distance. The trajectory can be absolute or relative.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Block Number							
2	0	0	1	Command Message Type (00001)				
3	0	0	1	Response Message Type				
4	Target Position Low Byte							
5	Target Position Low Middle Byte							
6	Target Position High Middle Byte							
7	Target Position High Byte							

Enable Setting this bit enables the drive. Also see Enable attribute (attribute 17), Position Controller Object Class (ID=37).

Registration Arm This bit is used to arm the registration input. When the registration input is triggered, the registration action will be executed. Also see Registration Arm attribute (attribute 21), Position Controller Supervisor Object Class (ID=36).

Hard Stop Setting this bit causes the drive to stop immediately (without decelerating). Also see Hard Stop attribute(attribute 21), Position Controller Object Class (ID=37)**.**

Smooth Stop Setting this bit causes the drive to decelerate to a stop. Also see Smooth Stop attribute (attribute 20), Position Controller Object Class (ID=37)**.**

NOTE

To stop motion, issue either HARD STOP or SMOOTH STOP only. Changing either one of these bits at the same time as the Start Trajectory bit causes indeterminate action from the controller.

Direction This bit is used only in velocity mode. It is used to instantaneously change the direction of travel. 1 – forward, positive and 0 – reverse, negative. Also see Direction attribute (attribute 23), Position Controller Class Object Class (ID=37). Used only with Command Assembly 2.

Incremental This bit is used in only in position mode. This bit indicates whether the position specified in bytes 4 through 7 of the Command Assembly 1 – Target Position, is absolute (0) or incremental (1). See the description for Incremental Mode Flag attribute (attribute 10) , Position Controller Object Class (ID=37).

Start Block This bit is used to execute a Command Block or Command block chain. Set from zero to one to execute a command block or command block chain. A number of command block exists in Block Number (byte one of Command Message). It will be valid only if Load Data / Start Profile bit is zero.

Load Data / Start Profile Set from zero to one to load data. The transition of this bit from zero to one will also start a Profile Move when the command message type contained in the command message field is the message type that starts a Profile Move for the mode selected.

Block Number This byte defines the block number to be executed when the Start Block bit transitions from zero to one.

Command Message Type This field defines the Command Message Type

Response Message Type This field defines the Response Message Type

Target Position This double word defines the Profile Move's Target Position in position units, when the Load Data / Start Profile bit transitions from zero to one.

Shown below is an example for the SERVO**STAR** S- and SERVO**STAR** CD-Series drives. Since the device has a motor resolution of 12800 position units per revolution, and in this example, the target position is set to 10 revs (or 128000 position units, or in hexadecimal, 0x0001F400 position units). The enable bit is set, along with the incremental bit and the start bit. Lastly, this example indicates that the desired Polled I/O Response Assemblies is the actual velocity (refer to Response Assembly 3 – Actual Velocity for more information).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	0	0	1	0	1
1	0							
2	0	0	1	0	0	0	0	1
3	0	0	1	0	0	0	1	1
4	0x00							
5	0xF4							
6	0x01							
7	0x00							

NOTE

You must make sure the transition of Trajectory Start is seen by the drive. The scan time of the system determines if the drive sees the change in bit 0. You may have to wait one scan cycle or more to make sure the change is seen by the drive.

Below is an example you may wish to use to determine if the drive has accepted the new command. In this example, we wish to change the Load Data / Start Profile bit. By changing the Response Assembly at the same time, the drive is forced to respond with a message that may not correspond to the actual values in the system

Example	*Command*	*Response*	*Notes*
Current Value	*0x80 0x00 0x21 0x21*	*0x94 0x00 0x00 0x21*	*The current value is shown*
Desired Value	***0x81*** *0x00 0x21* ***0x22***	*0x94 0x00 0x00 0x21*	*Desired value shown in bold, along with bit to indicate change*
Next Cycle	***0x81*** *0x00 0x21* ***0x22***	*0x94 0x00 0x00 0x21*	*Trajectory Start bit not seen yet because the last byte stay the same*
Next Cycle	*0x81 0x00 0x21 0x22*	*0x94 0x00 0x00 0x22*	*Trajectory Start bit seen because the last byte changed.*

Command Assembly 2 – Target Velocity

This command assembly is used to change the target velocity (position or velocity mode) and start trajectory (velocity mode only).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Block Number							
2	0	0	1	Command Message Type (00010)				
3	0	0	1	Response Message Type				
4	Target Velocity Low Byte							
5	Target Velocity Low Middle Byte							
6	Target Velocity High Middle Byte							
7	Target Velocity High Byte							

See Command Assembly 1: Target Position for bit descriptions.

Target Velocity This double word defines the Profile Move's Target Velocity in profile units, when the Load Data / Start Profile bit transitions from zero to one.

Shown below is an example for the SERVO**STAR** S- and SERVO**STAR** CD-Series drives. Since the device has a motor resolution of 12800 position units per revolution, and in this example, the target velocity is set to 20 RPS (or 256000 position units/sec, or in hexadecimal, 0x0003e800 position units/sec). The enable bit is set, along with the start trajectory (which instructs the drive to change the target velocity). In velocity mode, the drive immediate accelerates or decelerates to 20 RPS. In position mode, the next trajectory has a target velocity of 20 RPS. Lastly, this example indicates that the desired Polled I/O Response Assemblies is the actual position (refer to Response Assembly 1 – Actual Position for more information).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	0	0	0	0	1
1	0							
2	0	0	1	0	0	0	1	0
3	0	0	1	0	0	0	0	1
4	0x00							
5	0xe8							
6	0x03							
7	0x00							

Command Assembly 3 – Acceleration

This command assembly is used to change the acceleration.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Block Number							
2	0	0	1	Command Message Type (00011)				
3	0	0	1	Response Message Type				
4	Acceleration Low Byte							
5	Acceleration Low Middle Byte							
6	Acceleration High Middle Byte							
7	Acceleration High Byte							

See Command Assembly 1: Target Position for bit descriptions.

Acceleration This double word defines the Profile Move's Acceleration in profile units, when the Load Data / Start Profile bit transitions from zero to one.

Shown below is an example for the SERVO**STAR** S- and SERVO**STAR** CD-Series drives. Since the device has a motor resolution of 12800 position units per revolution, and in this example, the acceleration is set to 20 rps^2 (or 256000 position units/sec^2, or in hexadecimal, 0x0003e800 position units/ sec^2). The enable bit is set, along with the start trajectory (which instructs the drive to change the acceleration). Lastly, this example indicates that the desired Polled I/O Response Assemblies is the actual position (refer to Response Assembly 1 – Actual Position for more information).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	0	0	0	0	1
1	0							
2	0	0	1	0	0	0	1	1
3	0	0	1	0	0	0	0	1
4	0x00							
5	0xe8							
6	0x03							
7	0x00							

Command Assembly 4 – Deceleration

This command assembly is used to change the deceleration. This can only be used in position or velocity mode.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Block Number							
2	0	0	1	Command Message Type (00100)				
3	0	0	1	Response Message Type				
4	Deceleration Low Byte							
5	Deceleration Low Middle Byte							
6	Deceleration High Middle Byte							
7	Deceleration High Byte							

See Command Assembly 1: Target Position for bit descriptions.

Deceleration This double word defines the Profile Move's Deceleration in profile units, when the Load Data / Start Profile bit transitions from zero to one.

Shown below is an example for the SERVO**STAR** S- and SERVO**STAR** CD-Series drives. Since the device has a motor resolution of 12800 position units per revolution, and in this example, the deceleration is set to 20 rps^2 (or 256000 position units/sec^2, or in hexadecimal, 0x0003e800 position units/ sec^2). The enable bit is set, along with the start trajectory (which instructs the drive to change the deceleration). Lastly, this example indicates that the desired Polled I/O Response Assemblies is the actual position (refer to Response Assembly 1 – Actual Position for more information).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	0	0	0	0	1
1	0							
2	0	0	1	0	0	1	0	0
3	0	0	1	0	0	0	0	1
4	0x00							
5	0xe8							
6	0x03							
7	0x00							

Command Assembly 26 – Position Controller Supervisor Attribute

This command assembly is used to access and set any attributes in the Position Controller Supervisor group. The command assembly allows a Set and Get function to be performed simultaneously.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Position Controller Supervisor Attribute to Get							
2	0	0	1	Command Message Type (26)				
3	Position Controller Supervisor Attribute to Set							
4	Position Controller Supervisor Attribute Value Low Byte							
5	Position Controller Supervisor Attribute Value Low Middle Byte							
6	Position Controller Supervisor Attribute Value High Middle Byte							
7	Position Controller Supervisor Attribute Value High Byte							

Position Controller Supervisor to Get determines the response from the Response Assembly. Position Controller Supervisor Attribute to Set determines which attribute to set. Command Message Type can either be 26 or 27. The value determines if the drive is setting the Position Controller Supervisor or the Position Controller.

In the following example, the Registration Action (Class, 36, Attribute 19) is set to Hard Stop. Furthermore, the drive is instructed to respond with the axis number (Class 36, Attribute 3).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	03							
2	0	0	1	26				
3	19							
4	0x01							
5	0x00							
6	0x00							
7	0x00							

Command Assembly 27 – Position Controller Attribute

This assembly allows any attribute in the Position Controller group to be set or accessed. The command assembly allows a Set and Get function to be performed simultaneously.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Position Controller Attribute to Get							
2	0	0	1	Command Message Type (27)				
3	Position Controller Attribute to Set							
4	Position Controller Attribute Value Low Byte							
5	Position Controller Attribute Value Low Middle Byte							
6	Position Controller Attribute Value High Middle Byte							
7	Position Controller Attribute Value High Byte							

In the following example, the Incremental Mode Flag (Class 37, Attribute 10) is set to high (1), which causes the drive to move incrementally. In response, the drive is commanded to return the current Target Position (Class 37, Attribute 6).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	06							
2	0	0	1	27				
3	10							
4	0x01							
5	0x00							
6	0x00							
7	0x00							

Command Assembly 28 – Block Sequencer Attribute

This assembly allows any attribute in the Block Sequencer group to be set or accessed. The command assembly allows a Set and Get function to be performed simultaneously.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Block Sequencer Attribute to Get							
2	0	0	1	Command Message Type (28)				
3	Block Sequencer Attribute to Set							
4	Block Sequencer Attribute Value Low Byte							
5	Block Sequencer Attribute Value Low Middle Byte							
6	Block Sequencer Attribute Value High Middle Byte							
7	Block Sequencer Attribute Value High Byte							

In the following example, the Counter (Class 38, Attribute 6) is set 5, which means that will be 5 sequencer loops, and here we want also to read Block execution flag (attribute 2).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Dir (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	02							
2	0	0	1	28				
3	06							
4	0x05							
5	0x00							
6	0x00							
7	0x00							

Command Assembly 30 – Command Block Attribute

This assembly allows any attribute and any instance in the Command Block group to be set or accessed. The command assembly allows a Set and Get function to be performed simultaneously.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	Command Block Attribute to Get/Set							
2	0	0	1	Command Message Type (30)				
3	Command Block Instance to Get/Set							
4	Command Block Attribute Value Low Byte							
5	Command Block Attribute Value Low Middle Byte							
6	Command Block Attribute Value High Middle Byte							
7	Command Block Attribute Value High Byte							

In the following example, the Block Link Number (Class 39, Attribute 2 Instance 5) is set to 8, which causes that after executed block number 5 will be executed block number 8.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Arm	Hard Stop	Smooth Stop	Direction (Vel Mode)	Incremental	Start Block	Load Data / Start Profile
1	02							
2	0	0	1	30				
3	05							
4	0x08							
5	0x00							
6	0x00							
7	0x00							

Polled I/O Response Assemblies

Response Assembly 1 – Actual Position

This response assembly is used to return the Actual Position of the motor (in position units).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Executing Block Number							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (00001)				
4	Actual Position Low Byte							
5	Actual Position Low Middle Byte							
6	Actual Position High Middle Byte							
7	Actual Position High Byte							

Enable This bit reflects the enable state of the drive.
See the description of Enable attribute (Class 37 , Attribute 17).

Registration Level This bit reflects the level of the Registration Input of the drive (it this input is defined or 0 if not defined). See the description of Registration Input Level attribute (Class 36, Attribute 22).

Home Level This bit reflects the level of the Home Input of the drive. See the description of Home Level attribute (Class 36, Attribute 16).

Current Direction This bit reflects the direction of motion.
See the description of Direction attribute (Class 37, Attribute 23).

General Fault This bit indicates whether or not a fault has occurred.
See the description of General Fault attribute (Class 36, Attribute 5).

On Target Position This bit indicates whether or not the motor is on the last targeted position (1-On Target). See the description for Incremental Position Flag attribute (Class 37, Attribute 12).

Block in Execution When set, indicates drive is running a program. See the description of Block Execute attribute (Class 38, Attribute 2).

Profile in Progress This bit indicates that a profile move is in progress.

Executing Block Number The byte defines the currently executing block if the Block In Execution bit is active. See the description of Current Block attribute (Class 38, Attribute 3).

Load Complete This bit indicates that the command data contained in the command message has been successfully loaded into the device. See the description of Load Data Complete attribute (Class 37, Attribute 58).

Block Fault This bit indicates that a block execution fault occurred. See the description of Block Fault attribute (Class 38, Attribute 4).

Following Error This bit indicates when a following (static or dynamic) error occurs. See the description of Following Error Fault attribute (Class 37, Attribute 47).

Negative Limit	This bit indicates when the position is less than or equal to the Negative Soft Limit Position. See the description for Negative Soft Limit (Class 37 – Position Controller, Attribute 57).
Positive Limit	This bit indicates when the position is less than or equal to the Positive Soft Limit Position. See the description for Positive Soft Limit (Class 37 – Position Controller, Attribute 56).
CCW Limit	This bit indicates the state of the CCW Limit Input. See the description for CCW Limit (Class 37 – Position Controller, Attribute 51).
CW Limit	This bit indicates the state of the CW Limit Input. See the description for CW Limit (Class 37 – Position Controller, Attribute 50).
Actual Position	The double word reflects the actual position in position units.

In the example below, the device has a motor revolution of 12,800 position units per revolution and the actual position is 10 revolutions (or 128,000 position units, or in hexadecimal, 0x0001F400 position units). The enable bit is set, the Registration Level is 0, the Home Level is 0, the direction is positive (1), there are no faults (0), we are On Target Position (1), no Block in Execution (0) and Profile in Progress, no Block Fault (0), no Following Error (0), not on Negative Limit (0), on Positive Limit (1), not on CCW Limit (0), on CW Limit (1) and the Fault Input is not active (0).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1	0							
2	0	0	0	0	1	0	1	0
3	0	0	1	0	0	0	0	1
4	0x00							
5	0xF4							
6	0x01							
7	0x00							

Response Assembly 2 – Commanded Position

This response assembly is used to return the commanded position of the motor (in position units).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Dir	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Executing Block Number							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (00010)				
4	Commanded Position Low Byte							
5	Commanded Position Low Middle Byte							
6	Commanded Position High Middle Byte							
7	Commanded Position High Byte							

Commanded Position The double word reflects the commanded or calculated position in position units.

In the example below, the device has a motor resolution of 12,800 position units per revolution and the commanded position is 10 revolutions (or 128,000 position units, or in hexadecimal, 0x0001F400 position units). The enable bit is set, the Registration Level is 0, the Home Level is 0, the direction is positive (1), there are no faults (0), we are On Target Position (1), no Block in Execution (0), Profile in Progress, no Block Fault (0), no Following Error (0), not on Negative Limit (0), on Positive Limit (1), not on CCW Limit (0), not on CW Limit (1) and the Fault Input is not active (0).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1	0							
2	0	0	0	0	1	0	0	0
3	0	0	1	0	0	0	1	0
4	0x00							
5	0xF4							
6	0x01							
7	0x00							

Response Assembly 3 – Actual Velocity

This response assembly returns the actual velocity of the motor (in position units/sec).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Executing Block Number							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (00011)				
4	Actual Velocity Low Byte							
5	Actual Velocity Low Middle Byte							
6	Actual Velocity High Middle Byte							
7	Actual Velocity High Byte							

Actual Velocity The double word reflects the actual velocity in profile units.

In the example below, the device has a motor resolution of 12,800 position units per revolution and the actual velocity is 10 revs/sec (or 128,000 position units/sec, or in hexadecimal, 0x0001F400 position units/sec). The enable bit is set, the Registration Level is 0, the Home Level is 0, the direction is positive (1), there are no faults (0), we are On Target Position (1), no Block in Execution (0), Profile in Progress, no Block Fault (0), no Following Error (0), not on Negative Limit (0), on Positive Limit (1), not on CCW Limit (0), not on CW Limit (1) and the Fault Input is not active (0).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1	0							
2	0	0	0	0	1	0	0	0
3	0	0	1	0	0	0	1	1
4	0x00							
5	0xF4							
6	0x01							
7	0x00							

Response Assembly 4 – Commanded Velocity

This response assembly returns the commanded velocity of the motor (in position units/sec).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Dir	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Executing Block Number							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (00100)				
4	Commanded Velocity Low Byte							
5	Commanded Velocity Low Middle Byte							
6	Commanded Velocity High Middle Byte							
7	Commanded Velocity High Byte							

Commanded Velocity The double word reflects the commanded or calculated velocity in profile units.

In the example below, the device has a motor resolution of 12,800 position units per revolution and the commanded velocity is 10 revs/sec (or 128,000 position units/sec, or in hexadecimal, 0x0001F400 position units/sec). The enable bit is set, the Registration Level is 0, the Home Level is 0, the direction is positive (1), there are no faults (0), we are On Target Position (1), no Block in Execution (0), Profile in Progress, no Block Fault (0), no Following Error (0), not on Negative Limit (0), on Positive Limit (1), not on CCW Limit (0), not on CW Limit (1) and the Fault Input is not active (0).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1	0							
2	0	0	1	0	0	1	0	0
3	0	0	1	0	0	1	0	0
4	0x00							
5	0xF4							
6	0x01							
7	0x00							

Response Assembly 6 – Home Position

This response assembly returns the home position of the motor (in position units).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Dir	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Executing Block Number							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (00110)				
4	Home Position Low Byte							
5	Home Position Middle Byte							
6	Home Position High Middle Byte							
7	Home Position High Byte							

Home Position This double word reflects the captured home position in position units.

In the example below, the device has a motor resolution of 12,800 position units per revolution and the home position is 0 revs (or 0 position units, or in hexadecimal, 0x00000000 position units). The enable bit is set, the Registration Level is 0, the Home Level is 0, the direction is positive (1), there are no faults (0), we are On Target Position (1), no Block in Execution (0), Profile in Progress, no Block Fault (0), no Following Error (0), not on Negative Limit (0), on Positive Limit (1), not on CCW Limit (0), not on CW Limit (1) and the Fault Input is not active (0).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1	0							
2	0	0	0	0	1	0	0	0
3	0	0	1	0	0	1	1	0
4	0x00							
5	0x00							
6	0x00							
7	0x00							

Response Assembly 8 – Registration Position

This response assembly returns the index position of the motor (in position units).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Executing Block Number							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (01000)				
4	Registration Position Low Byte							
5	Registration Position Middle Byte							
6	Registration Position High Middle Byte							
7	Registration Position High Byte							

Registration Position This double word reflects the captured registration position in position units.

In the example below, the device has a motor resolution of 12,800 position units per revolution and the registration position is 10 revs (or 128,000 position units, or in hexadecimal, 0x0001F400 position units). The enable bit is set, the Registration Level is 0, the Home Level is 0, the direction is positive (1), there are no faults (0), we are On Target Position (1), no Block in Execution (0), Profile in Progress, no Block Fault (0), no Following Error (0), not on Negative Limit (0), on Positive Limit (1), not on CCW Limit (0), not on CW Limit (1) and the Fault Input is not active (0).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1	0							
2	0	0	0	0	1	0	0	0
3	0	0	1	0	1	0	0	0
4	0x00							
5	0xF4							
6	0x01							
7	0x00							

Response Assembly 20 – Command/Response Error

This response assembly returns the information about error that existed in command or response message.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Reserved = 0							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (10100)				
4	General Error Code							
5	Additional Code							
6	Copy of Command Message Byte 2							
7	Copy of Command Message Byte 3							

General Error Code This byte identifies an error has been encountered.

The table below gets more information about general error code :

General Error Code	Additional Code	Response	Description
0x08	0x01	Service Not Supported	Command Message type not supported.
0x08	0x02	Service Not Supported	Response Message type not supported.
0x09	0xFF	Invalid Attribute Value	Load value is out of range.
0x0E	0xFF	Attribute not Settable	A request to modify a non-modifiable attribute was received.
0x13	0xFF	Not Enough Data	I/O Command message contained fewer than 8 bytes.
0x14	0xFF	Attribute Not Supported	Attribute specified in request was not supported

Additional Code This byte contains an object/service- specific value that further describes the error condition. If the responding object has no additional information to specify, then the value 0xFF is placed within this field.

In the example below, the device has a motor resolution of 12,800 position units per revolution and the registration position is 10 revs (or 128,000 position units, or in hexadecimal, 0x0001F400 position units). The enable bit is 0, the Registration Level is 0, the Home Level is 0, the direction is positive (1), there are faults (1), we are On Target Position (1), no Block in Execution (0), Profile in Progress, no Following Error (0), not on Negative Limit (0), on Positive Limit (1), not on CCW Limit (0), not on CW Limit (1) and the Fault Input is not active (0).

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	0	1	0	1
1	0							
2	0	0	0	0	1	0	0	0
3	0	0	1	1	0	1	0	0
4	0x08							
5	0x01							
6	0x3f							
7	0x01							

Response Assembly 26 – Position Controller Supervisor Attribute

This response assembly is used to access the current state of the Position Controller Supervisor Attributes without using Explicit Messaging. The last four bytes of the assembly is the current value of the attribute.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Position Controller Supervisor Attribute to Get							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (26)				
4	Position Controller Supervisor Attribute Value Low Byte							
5	Position Controller Supervisor Attribute Value Middle Byte							
6	Position Controller Supervisor Attribute Value High Middle Byte							
7	Position Controller Supervisor Attribute Value High Byte							

The example below shows the response from the drive for Class 36, Attribute 24, the Registration position. With a feedback resolution of 4096, the current setting indicates that the Registration Position is 1 rev from the registration input. The drive is enabled and was moving in the clockwise direction. Finally, the motor is in position.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1					24			
2	0	0	0	0	0	0	0	0
3	0	0	1	26				
4	0x00							
5	0x10							
6	0x00							
7	0x00							

Response Assembly 27 – Position Controller Attribute

This response assembly is used to access the current state of the Position Controller Attributes without using Explicit Messaging. The last four bytes of the assembly are the current value of the attribute.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Position Controller Attribute to Get							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (27)				
4	Position Controller Attribute Value Low Byte							
5	Position Controller Attribute Value Middle Byte							
6	Position Controller Attribute Value High Middle Byte							
7	Position Controller Attribute Value High Byte							

The example below shows the response from the drive for Class 37, Attribute 40, the Feedback Resolution. The response indicates that the feedback resolution is set at 4096. It further shows that the drive is enabled and was moving in the clockwise direction. The motor is also in position.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1					40			
2	0	0	0	0	0	0	0	0
3	0	0	1	27				
4	0x00							
5	0x10							
6	0x00							
7	0x00							

Response Assembly 28 – Block Sequencer Attribute

This response assembly is used to access the current state of the Block Sequencer Attributes without using Explicit Messaging. The last four bytes of the assembly are the current value of the attribute.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Block Sequencer Attribute to Get							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (28)				
4	Block Sequencer Attribute Value Low Byte							
5	Block Sequencer Attribute Value Middle Byte							
6	Block Sequencer Attribute Value High Middle Byte							
7	Block Sequencer Attribute Value High Byte							

The example below shows the response from the drive for Class 38, Attribute 3, the Current Block. According to the Value we can see current block in execution. In example we see that current block in execution is 3.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1					3			
2	0	0	0	0	0	0	0	0
3	0	0	1	28				
4	0x03							
5	0x00							
6	0x00							
7	0x00							

Response Assembly 30 – Command Block Attribute

This response assembly is used to access the current state of the Command Block Attributes without using Explicit Messaging. The last four bytes of the assembly are the current value of the attribute.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Enable	Reg Level	Home Level	Current Direction	General Fault	On Target Position	Block in Execution	Profile in Progress
1	Command Block Attribute to Get							
2	Load Compl.	Block Fault	Following Error	Negative Limit	Positive Limit	CCW Limit	CW Limit	Fault Input Active
3	0	0	1	Response Message Type (30)				
4	Command Block Attribute Value Low Byte							
5	Command Block Attribute Value Middle Byte							
6	Command Block Attribute Value High Middle Byte							
7	Command Block Attribute Value High Byte							

The example below shows the response from the drive for Class 39, Attribute 2, the Block Link Number. According to the Value we can see Block link instance number. In example we see that block link instance number is 3.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	1	0	0	1	0	1	0	1
1					2			
2	0	0	0	0	0	0	0	0
3	0	0	1	30				
4	0x03							
5	0x00							
6	0x00							
7	0x00							

Identity Object Class (0x01)

Class Attributes

Id	Description	Get	Set	Limits
1	Revision	-	-	
2	Max Instance	-	-	
3	Number of instances	-	-	
4	Optional Attribute List	-	-	
5	Optional Service List	-	-	
6	Max ID of class attributes	-	-	
7	Max ID of instance attributes	-	-	

Class Services

Service		Parameter Options
Get_Attributes_All	-	
Reset	-	
Get_Attribute_Single	-	
Find_Next_Object_Instance	-	

Instance Attributes

Id	Description	Get	Set	Limits
1	Vendor	+	-	
2	Device Type	+	-	
3	Product Code	+	-	
4	Revision	+	-	
5	Status	+	-	
6	Serial Number	+	-	
7	Product Name	+	-	
8	State	-	-	
9	Configuration Consistency Value	+		
10	Heartbeat Interval	+	+	

Instanse Services

Service		Parameter Options
Get_Attributes_All	-	
Reset	+	
Get_Attribute_Single	+	
Set_Attribute_Single	+	

Message Router Class (0x02)

Class Attributes

Id	Description	Get	Set	Limits
1	Revision	-	-	
2	Max Instance	-	-	
3	Number of instances	-	-	
4	Optional Attribute List	-	-	
5	Optional Service List	-	-	
6	Max ID of class attributes	-	-	
7	Max ID of instance attributes	-	-	

Class Services

Service		Parameter Options
Get_Attributes_All	-	
Get_Attribute_Single	-	

Instance Attributes

Id	Description	Get	Set	Limits
1	Object List	-	-	
2	Maximum connections supported	-	-	
3	Number of active connections	-	-	
4	Active connections list	-	-	

Instanse Services

Service		Parameter Options
Get_Attributes_All	-	
Get_Attribute_Single	+	

DeviceNet Object (0x03)

Class Attributes

Id	Description	Get	Set	Limits
1	Revision	+	-	
2	Max Instance	-	-	
3	Number of instances	-	-	
4	Optional Attribute List	-	-	
5	Optional Service List	-	-	
6	Max ID of class attributes	-	-	
7	Max ID of instance attributes	-	-	

Class Services

Service		Parameter Options
Get_Attribute_Single	+	

Instance Attributes

Id	Description	Get	Set	Limits
1	MAC ID	+	+*	
2	Baud Rate	+	+*	
3	BOI	-	-	
4	Bus-off counter	-	-	
5	Allocation information	+	-	
6	MAC ID switch changed	-	-	
7	Baud rate switch changed	-	-	
8	MAC ID switch value	-	-	
9	Baud rate switch value	-		

* Only settable if non-volatile network parameters are used (no DIP switch)

Instanse Services

Service		Parameter Options
Get_Attribute_Single	+	
Set_Attribute_Single	+*	
Allocate M/S connection set	+	
Release M/S connection set	+	

* Only available if non-volatile parameters are used (no DIP switch)

Assembly Object (0x04)

Class Attributes

Id	Description	Get	Set	Limits
1	Revision	-	-	
2	Max Instance	-	-	
3	Number of instances	-	-	
4	Optional Attribute List	-	-	
5	Optional Service List	-	-	
6	Max ID of class attributes	-	-	
7	Max ID of instance attributes	-	-	

Class Services

Service		Parameter Options
Create	-	
Delete	-	
Get_Attribute_Single	-	

Instance Attributes

Id	Description	Get	Set	Limits
1	Number of members	-	-	
2	Member List	-	-	
3	Data	+	+	For Output and Cfg objects only

Instanse Services

Service		Parameter Options
Delete	-	
Get_Attribute_Single	+	
Set_Attribute_Single	+*	
Get_Member	-	
Insert_Member	-	
Remove_Member	-	

* Only available for Output and Configuration objects

Connection Object (0x05)

Class Attributes

Id	Description	Get	Set	Limits
1	Revision	-	-	
2	Max Instance	-	-	
3	Number of instances	-	-	
4	Optional Attribute List	-	-	
5	Optional Service List	-	-	
6	Max ID of class attributes	-	-	
7	Max ID of instance attributes	-	-	

Class Services

Service		Parameter Options
Reset	-	
Create	-	
Delete	-	
Get_Attribute_Single	-	
Find_Next_Object_Instance	-	

Instance Attributes

Id	Description	Get	Set	Limits
1	State	+	-	
2	Instance Type	+	-	
3	Transport class trigger	+	-	
4	Produced connection ID	+	-	
5	Consumed connection ID	+	-	
6	Initial comm. Characteristics	+	-	
7	Produced connection size	+	-	
8	Consumed connection size	+	-	
9	Expected packet rate	+	+	
12	Watchdog timeout action	+	+*	Settable to Deferred_Delete or Auto_Delete only
13	Produced connection path length	+		
14	Produced connection path	+	+**	
15	Consumed connection path length	+		
16	Consumed connection path	+	+***	
17	Production Inhibit Time	+	+****	

* Only settable for explicit messaging connection instances
** For Poll/Strobe/COS/Cyclic I/O connection instances
*** For Poll/Strobe connection instances
**** Only settable for COS/Cyclic I/O connection instances

Instanse Services

Service		Parameter Options
Reset	-	
Delete	-	
Apply_Attributes	-	
Get_Attribute_Single	+	
Set_Attribute_Single	+	

Ack Handler Object (0x2B)

Class Attributes

Id	Description	Get	Set	Limits
1	Revision	-	-	
2	Max Instance	-	-	
3	Number of instances	-	-	
4	Optional Attribute List	-	-	
5	Optional Service List	-	-	
6	Max ID of class attributes	-	-	
7	Max ID of instance attributes	-	-	

Class Services

Service		Parameter Options
Create	-	
Delete	-	
Get_Attribute_Single	-	

Instance Attributes

Id	Description	Get	Set	Limits
1	Acknowledge Timer	+	+	
2	Retry Limit	+	-	
3	COS Producing Connection Inst	+	-	
4	Ack List Size	-	-	
5	Ack List	-	-	
6	Data with Ack Path List Size	-	-	
7	Data with Ack Path List	-	-	

Instanse Services

Service		Parameter Options
Delete	-	
Get_Attribute_Single	+	
Set_Attribute_Single	+	