KOLLMORGEN

www.DanaherMotion.com

# SERVOSTAR® MC


**M-SS-005-03**
**Revision E**
**Firmware Version 5.0.0**

DANAHER MOTION

**Record of Manual Revisions**

| Revision | Date | Description of Revision |
|---|---|---|
| 0 | 3/1/1999 | Preliminary issue for review |
| 1 | 5/14/1999 | Initial release |
| 2 | 6/5/2000 | New PCI and stand-alone models, new firmware features |
| 3 | 8/31/2001 | New firmware features |
| D | 5/11/2005 | Update examples |
| E | 6/20/2005 | Updated firmware version and dates |



Safety-alert symbols used in this document are:

**WARNING**

***Warnings alert users to potential physical danger or harm. Failure to follow warning notices could result in personal injury or death.***

**CAUTION**

***Cautions direct attention to general precautions, which if not followed, could result in personal injury and/or equipment damage.***

**NOTE**

***Notes highlight information critical to your understanding or use of the product.***

# Table of Contents

# 1. OVERVIEW

This manual describes how to use the SERVO**STAR**® MC (**M**ulti-axis **C**ontroller) product. To execute the examples described in this manual, you must at least have a SERVO**STAR** MC and BASIC Moves Development Studio® installed on your host computer. Some examples further require that you have a drive installed and connected to the MC using the SERCOS Interface™ cables. The *SERVOSTAR® MC Installation Manual* describes the procedures for all the necessary installations.

Version 5.0.0 of the firmware incorporates many new commands, functions, and properties, and, in some instances, the behavior of functions, commands, and properties found in previous versions of the firmware have been changed. This edition of the *SERVO**STAR®** MC User's Manual* applies to version 5.0.0 firmware, and is not necessarily backward compatible with previous versions of the firmware. If you have used previous versions of the MC, you may find that some functions, commands, or properties which you are familiar with, now have different behavior, or may no longer exist. Refer to the *SERVO**STAR®** MC Reference Manual* for additional information, as it covers all versions of the firmware.

When you complete this manual you should feel comfortable using all the features available in the SERVO**STAR** MC. Related manuals:

*SERVO**STAR®** MC Installation Manual*
*SERVO**STAR®** MC BASIC Moves Development Studio Manual*
*SERVO**STAR®** SC/MC API Reference Manual*
*SERVO**STAR®** MC InstallationManual*

The SERVO**STAR** MC is Danaher Motion's multi-axis controller. The MC has the features and performance required of today's multi-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**
Danaher Motion can provide a complete motion control system,including controller, drives, and motors. All the components are supplied by DanaherMotion, and are tested and guaranteed to work together.

**Complete Digital System at an Affordable Price**
The MC is fully digital, including communication with the drives. The analog interface is eliminated, but the price is competitive with analog systems.

**Local Field Bus for Simple Set up, Less Wiring, More Flexibility**
The MC communicates via fiber-optic cables using an industrial field bus called SERCOS. SERCOS greatly simplifies system set-up and wiring, while improving system flexibility and reliability compared to its analog counterparts.

**Windows®-Based Software Supports Microsoft Visual Basic™, Visual C++™, and other Popular Languages**
The MC product family includes Danaher Motion's Kollmorgen API, which allows Windows NT® application integration with the MC controller. Communication uses dual-port RAM, so the process is fast and reliable.

# 1. 1 SYSTEM HARDWARE

The MC hardware is available in two types of implementation: a plug-in circuit board for a host computer, and a stand-alone model.

The PCI plug-in board hardware installs in a host computer (typically a PC), running the Windows NT 4.0, Windows 2000/XP operating systems. The MC contains a fully-independent computer system., so it does not compete with the host processor system for resources. The MC uses Pentium 233 MHz or faster microprocessor to provide the power your system needs today, and the x86 upgrade path allows for future enhancement. The MC includes large on-board memory with ample capacity for your programs and enough space for expansion. There is also an onboard Flashdisk for storage of your programs.

The stand-alone model of the MC is designed for installation as a component part in industrial equipment. It consists of the PCI version plug-in board and a power supply in an enclosure. The stand-alone model does not have a host CPU to provide a Windows operating system and environment, so it can not directly provide an operator interface. For interactive operation of the stand-alone MC, you will need a host computer running BASIC Moves Development Studio software and communicating with the MC *via* Ethernet or serial communications with Windows NT, Windows 2000/XP operating system. For additional information concerning communications configuration refer to the *SERVO**STAR**® MC Installation Manual*.

All implementation models of the SERVO**STAR** MC are functionally equivalent for servo motion control

# 1. 2 OPERATING SYSTEM

The SERVO**STAR** MC is based on the VxWorks® **R**eal**T**ime **O**perating **S**ystem (RTOS), providing a stable platform for the SERVO**STAR**'s software. VxWorks is produced by Wind River Systems, Incorporated. VxWorks is continuously evolving, providing a clear path for future upgrades to the MC.

# 1. 3 MC-BASIC LANGUAGE COMPILER

The MC uses a language interpreter that semi-compiles commands so they typically execute in less than 5 microseconds. MC-BASIC® is true BASIC. It has familiar commands such as **FOR…NEXT**, **IF…THEN**, **PRINTUSING**, **PEEK** and **POKE** as well as common string functions such as **CHR$**, **INSTR**, **MID$**, and **STRING$**. It allows arrays (up to 10 dimensions) with full support for double precision floating-point math. MC-BASIC is extended to provide support for functions required for motion systems control, (*e.g.*, point to point moves, circular and linear interpolation, camming, and gearing). In addition, there is support for multi-tasking and event-driven programs.

Gearing and camming have the versatility required for motion control. You can slave any axis to any master. You can enable and disable gearing at any time. The gear ratio is expressed as a double-precision value. The MC also supports simulated axes, which can be master or slave in gearing and camming applications and incremental moves run by any axis, master or slave, at any time.

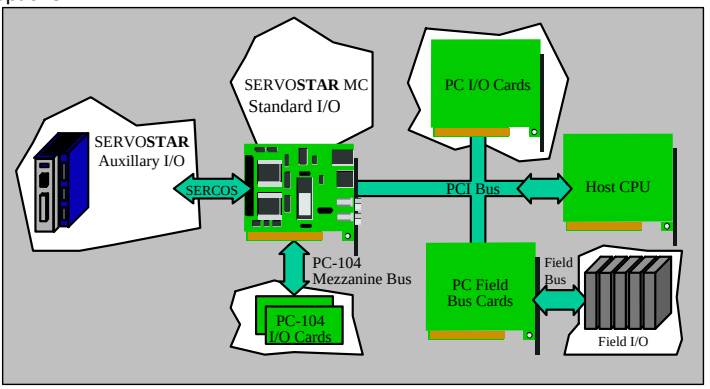
Camming links master and slave axes by a cam table. Camming has all the features of gearing, plus the cam table can have any number of points. The cam points are in x-y format so spacing can be provided independently. Multiple tables can be linked together, allowing you to build complex cam profiles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

The MC supports multi-tasking with up to 256 tasks, running at up to 16 different priority levels. 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 just about any combination of factors.

## 1. 4 I/O

The MC provides 23 optically-isolated inputs and 20 optically-isolated outputs as standard features, in addition to limit switches and other drive I/O points that are connected directly to the drives and transferred to the MC via SERCOS. Additionally, each SERVO**STAR** drive provides hardware for six I/O points: three digital inputs, one digital output, a 14-bit analog input, and a 12-bit analog output. An auxiliary encoder can also be connected to any SERVO**STAR** drive.

If you need more I/O, the MC includes a PC104 bus. Each MC can accommodate as many as two PC104 cards. You can also add I/O to your PC bus or other field buses such as DeviceNet and ProfiBus, and your application controls it all. The figure below shows the various possible I/O options.




## 1. 5 SERCOS

The SERCOS interface™ (developed in the mid-1980's) provides an alternative to the analog interface for communicating between motion controllers and drives. In 1995, SERCOS became an internationally accepted standard as IEC 1491 and later was continued to standard IEC 61491. The popularity of SERCOS has been steadily growing. All signals are transmitted between controller and drives on two fiber optic cables. This eliminates grounding noise and other types of Electro-Mechanical Conductance (EMC) and Electro-Mechanical Interference (EMI).

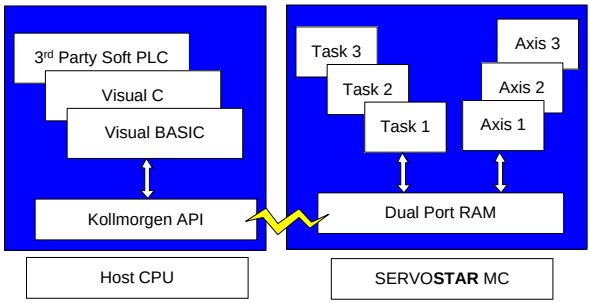
Because SERCOS is digital, it can transmit signals such as velocity and position commands with high resolution. The SERVO**STAR**® MC and accompanying drives ( SERVO**STAR** CD series), support 32-bit velocity commands. This provides a much higher resolution than can be achieved with the analog interface. The most common SERCOS functions are provided in such a way that you do not have to be an expert to gain the benefits.

## 1. 6 API

DanaherMotion's Kollmorgen API® is a software package that allows you to communicate with the SERVO**STAR** MC from popular programming languages, such as Visual Basic. The API provides complete access to all the elements of your system across dual-port ram. See the *SERVO**STAR**® SC/MC API Reference Manual* for more information.

## 1. 7 MULTI-TASKING

The SERVO**STAR** MC is a fully multi-tasking system (see figure below) in which you can create multiple tasks and multiple elements (e.g., axes) that operate independently of one another. Also, because the API supports multiple applications communicating concurrently, you can write one or more applications that have access to all of the tasks and elements.




## 1. 8 USER COMMUNICATION

The ETHERNET and Serial ports of the MC are used for ASCII data transfer. Non-printable characters are sent using **CHR$**. Data access is stream-oriented. There is no data framing. MC Basic applications gain access to either raw serial port or TCP socket. The TCP socket guaranties error free data transfer, while the serial port does not offer error recovery. The transmitted data does not have any meaning in terms of directly controlling the MC.

User communication provides the basic features of serial and TCP/IP data communication, which is not limited to a specific communication protocol, enabling the usage of any protocol over serial or TCP/IP through an MC application.

# 1.8.1. Serial Communication

The MC has two RS-232 ports, which can be used for user communication. When COM1 is used for communication with BASIC Moves over API, tasks cannot access it to eliminate interference with BMDS communications. Use DIPswitch bit #8 to configure this port:

| DIP Switch bit #8 state | |
|---|---|
| ON (default) | COM1 is reserved for communication with the host and cannot be used by any other application |
| OFF | **COM1 is available for user communication** |

When DIPswitch bit # 8 is ON, the MC's firmware prints boot status messages from COM1 using the following serial communication parameters:

| | |
|---|---|
| **Baudrate** | 9600 bps |
| **Parity** | none |
| **Stopbits** | 1 |
| **Databits** | 8 |

State ON of DIPswitch read as 0.

**NOTE**

For example:
Switches 6 & 8 are OFF. All others are ON.

```
-->?sys.dipswitch bin
0b10100000
-->
```

## *1.8.1.1. WITH HOST*

When DIPswitch #8 is ON, the MC can communicate with the host via serial port. This mode of communication requires Virtual Miniport driver, which is installed during setup of BMDS. The driver simulates Ethernet connection over serial interface using following fixed parameters:

| | |
|---|---|
| **Baudrate** | 115200 bps |
| **Parity** | None |
| **Stopbits** | 1 |
| **Databits** | 8 |
| **IP address** | 192.1.1.101 |
| **Subnet mask** | 255.255.255.0 |

All the parameters are fixed and cannot be modified.

If the driver is installed correctly and BMDS is running, it appears in the list of Ethernet adapters as shown below:

```
C:\>ipconfig /all
Ethernet adapter Local Area Connection 5:
Connection-specific DNS Suffix  . . . . :
Description . . . . . . . . . . . . . . : Giant Steps Virtual miniport
Physical Address. . . . . . . . . . . . : 00-02-1B-D1-A5-FC
DHCP Enabled. . . . . . . . . . . . . . : No
IP Address. . . . . . . . . . . . . . . : 192.1.1.1
Subnet Mask . . . . . . . . . . . . . . : 255.255.255.0
Default Gateway . . . . . . . . . . . . :
DNS Servers . . . . . . . . . . . . . . :
```

### 1.8.1.2. OPEN SERIAL PORT

Configurate the serial port with **OPEN**. Set the following port properties:

**BaudRate** - baud rate of the device set to a specified value.

**Parity** – enable/disable parity detection. When enabled, parity is odd or even.

**DataBits** - number of data bits.

**StopBit** - number of stop bits.

**Xonoff** – sets raw mode or ^S/^Q flow control protocol mode (optional parameter disabled by default).

For example:

```
OPEN COM2 BUADRATE=9600 PARITY=0 DATABITS=8 STOPBIT=1 AS #1
```

Opens COM2 at 9600 bps baud-rate, No parity, 8-bit data, 1 stop bit.

**OPEN** assigns a device handle to the communication port. Any further references to the communication port uses the device handle number. The device handle range is 1…255. To change the communication parameters, close the serial port and reopen it using the new parameters. For more information, see **PRINT #** and **INPUT$**.

## 1.8.2. TCP/IP Communication

Use TCP/IP communication with other computers and intelligent I/O devices over the Ethernet network. MC-BASIC uses TCP/IP API (sockets).

TCP/IP provides multi-thread communications. The same physical link (Ethernet) is shared between several applications. This way, the MC can connect to BASIC Moves and one or more intelligent I/O devices.

The MC supports both client and server roles for TCP/IP communications. The TCP/IP client connects to a remote system, which waits for a connection while the TCP/IP server accepts a connection from a remote system. Usually, the MC serves as client while interfacing to an intelligent I/O connected to the LAN, and as a server when working toward remote host like a software PLC. The MC uses Realtek RTL8019AS Ethernet NIC at a bit rate of 10 M bps.

### 1.8.2.1. SETUP

TCP communication starts by opening a socket either to connect to a remote computer (remote host) or accept a connection from a remote host. Use **CONNECT** or **ACCEPT**, depending on the MC functionality (client or server).

**OPENSOCKET** creates a TCP socket and assigns the socket descriptor to the specified device handle. The socket is created with **OPTIONS** NO_DELAY (send data immediately= 1). Otherwise, the data is buffered in the protocol stack until the buffer is full or a time-out is reached. This improves responsiveness when small amounts of data are sent. This option is recommended for interactive applications with relatively small data transfer. Otherwise, **OPTIONS** = 0.

```
OPENSOCKET OPTIONS=1 AS #1
```

**OPENSOCKET** assigns a device handle to the communication port (as **OPEN**). Any further reference to the communication port uses the device handle number. The device handle range is 1…255.

There are several ways to set the IP address of the controller. By default, the MC boots without a valid IP address. The following options are available to set the IP address of the controller:

**Static IP address setting**
Use SYS.IPADDRESSMASK in the Config.prg file to assign the IP address and Subnet mask:

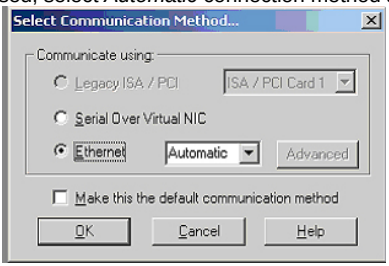
```
SYS.IPADDRESSMASK="212.25.84.109:255.255.255.128"
```

**Dynamic address setting by Windows API**
API assigns an IP address to the MC when it establishes TCP communication with the controller. The IP address and Subnet mask are taken out of a pre-defined address pool. BASIC Moves uses this address assigning method. Refer to the *BASIC Moves Development Studio® User Manual* and *SERVO**STAR**® MC Reference Manual* for detailed information.

**DHCP**
IP address is assigned by the DHCP server. If your network does not support DHCP, the controller tries to get the address from the DHCP server and times out after few minutes. During that time, communication with the controller is not possible.

```
SYS.IPADDRESSMASK="dhcp"
```

If DHCP is used, select *Automatic* connection method as shown below:




**Get the controller's IP address**
Use **SYS.IPADDRESSMASK** to query the IP address and Subnet mask of the MC.

```
-->?SYS.IPADDRESSMASK
172.30.3.11:255.255.0.0
```

**NOTE**

***The MC has several IP interfaces: Ethernet, IP over serial and IP over DPRAM. However** SYS.IPADDRESSMASK **is only valid for Ethernet interfaces. Others have fixed IP addresses, which cannot be changed.***

**PING** tests that a remote host is reachable. The remote host must support ICMP echo request.

```
?PING("212.25.84.109")
```

## 1.8.2.2. MC AS TCP CLIENT

When acting as a client, the MC tries to connect to a remote server until a time out value expires or the connection is rejected. Use **CONNECT** when the MC acts as a client.

The MC requests a connection from a remote host, according to a specific IP address and port. **CONNECT** blocks task execution until the connection is established or until the command fails. To unblock a task waiting in **CONNECT**, close the corresponding socket using **CLOSE**. In addition, killing a task (**KILLTASK**) blocked by **CONNECT** closes the socket to release the task. **CONNECT** may fail due to the following reasons:

1. Invalid socket descriptor
2. Remote host has no application attached to specified port.
3. Destination address is not reachable

The following example illustrates a typical procedure of establishing a connection to a remote host, and sending and receiving data:

```
OPENSOCKET OPTIONS=0 AS #1
CONNECT (#1,"212.25.84.100",6002)
   PRINT #1,"HELLO"
   ?INPUT$(LOC(1),#1)
CLOSE #1
```

### 1.8.2.3. MC AS TCP SERVER

When acting as a server. the MC endlessly waits for a connection from the remote host. Use **ACCEPT** when the MC acts as server. **ACCEPT** binds a socket to a specified port and waits for a connection. Simultaneous **ACCEPT**s on the same port are not allowed. **ACCEPT** blocks the execution of the calling task until a connection is established. To unblock the task waiting in **ACCEPT**, close the corresponding socket with **CLOSE**. In addition, killing a task (**KILLTASK**) blocked in **ACCEPT** closes the socket and releases the task. The following example illustrates a typical procedure of waiting for a connection from a remote host, and sending and receiving data:

```
OPENSOCKET OPTIONS=0 AS #1
ACCEPT (#1,20000)
   PRINT #1,"HELLO"
   ?INPUT$(LOC(1),#1)
CLOSE #1
```

## 1.8.3. Send /Receive Data

Serial and TCP communication use the same commands to send and receive data. After establishing communication over TCP or after configuring the serial port communication parameters, send and receive data regardless the communication type. Data flow differences are communication speed and the size of the input and output buffers. TCP/IP communication uses larger communication buffers.

### 1.8.3.1. RECEIVE DATA

The input-buffer of the serial ports is a fixed512 bytes size. User communication allows receiving strings using **INPUT$**. To check if data is ready at the input buffer, use **LOC**. The following example checks the number of bytes available at the input buffers and reads all the available data:

```
-->?LOC(1)
11
-->STRING_VAR = INPUT$(11, #1)
-->?STRING_VAR
Hello World
```

If **INPUT$** requests reading a data length larger than available data, the command returns only the available data:

```
-->?LOC(1)
11
-->STRING_VAR = INPUT$(20, #1)
-->?STRING_VAR
Hello World
```

**LOC** can be used within the **INPUT$**:

```
STRING_VAR = INPUT$(LOC(1), #1)
```

A partial read of the input buffer is allowed. This way, several calls to **INPUT$** empty the input-buffer:

```
-->?LOC(1)
11
-->STRING_VAR = INPUT$(3, #1)
-->?STRING_VAR
Hel
-->?LOC(1)
8
-->STRING_VAR = INPUT$(8, #1)
-->?STRING_VAR
lo World
```

When calling **INPUT$**, the system does not wait for data. If the input-buffer is empty, the command returns without any return value:

```
-->?LOC(1)
0
-->STRING_VAR = INPUT$(10, #1)
-->?len(STRING_VAR)
0
```

## 1.8.3.2. SEND DATA

**PRINT #** and **PRINTUSING #** send data over the serial port. Both commands use the same formatting as **PRINT** and **PRINTUSING**.

**NOTE**

*PRINT #* ***and*** *PRINTUSING #* ***appends a carriage-return (ASCII value 13) and line-feed (ASCII value 10) characters at the end of each message. To print a message without the terminating carriage-return and line-feed characters use a semicolon at the end of the command.***

For example, the following uses **PRINT #** to send data:

```
-->STRING_MESSAGE = "ERROR"
-->PRINT #1, STRING_MESSAGE,"A1.PCMD=";A1.PCMD;
```

The output is:

```
ERROR    A1.PCMD=0.000000000000000e+00
```

The values of the following variables are sent one after the other. The output is (hexadecimal) 0x37 0x28:

```
I1=55
I2=40
PRINT #1, I1;I2
```

The output is:

```
5540
```

## 1.8.3.3. SEND DATA BLOCK

**PRINTTOBUFF #** and **PRINTUSINGTOBUFF # a**llows buffering of data before actually being sent. This eliminates inter-character delays.

```
printtobuff #handle,chr$(0); chr$(message_length);
chr$(0);send=false
printtobuff #handle,chr$(request[ii]);send=false
printtobuff #handle,chr$(request[ii]);send=true
```

## 1.8.3.4. SEND/RECEIVE NULL CHARACTER

When using a specific protocol over Serial or TCP/IP ModBus RTU or ModBus TCP), the NULL character is part of the message. The message block cannot be saved in a STRING type variable since the NULL character terminates the string.

To send a NULL character in a message, send the data as single bytes instead of a string type message:

```
COMMON SHARED X[10] AS LONG
X[1]=0x10
X[2]=0
X[3]=0x10
PRINT #1, CHR$(X[1]);CHR$(X[2]);CHR$(X[3]);
```

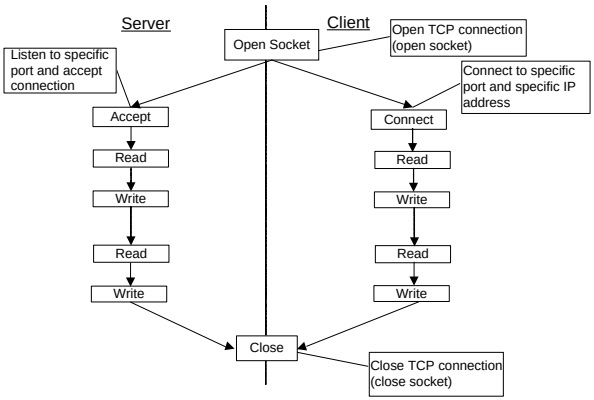
The output is the following stream of bytes:

```
0x10, 0x0, 0x10
```

When the received data contains a NULL character, read the data from the input buffer as single bytes instead of reading it into a string type variable. Convert the incoming data to a numeric value to get its ASCII value:

```
DIM SHARED TEMP[10] AS DOUBLE
FOR IDX = 1 TO LOC(1)
   TEMP[IDX] = ASC(INPUT$(1,#1))
NEXT IDX
```

The figure below gives a general description of the TCP/IP communication for both the client and host.




## 1.8.3.5. CLOSE CONNECTION

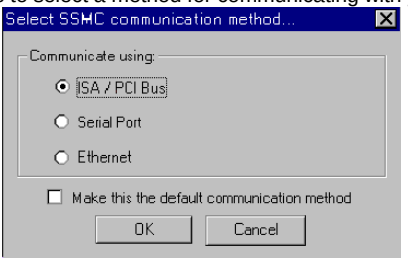
**CLOSE** closes the connection and releases the device handle:

```
-->CLOSE #1
```

# 2. BASIC MOVES DEVELOPMENT STUDIO

**B**ASIC **M**oves **D**evelopment **S**tudio (BMDS) provides Windows-based project control for each application. BASIC Moves Development Studio supports development for multi-tasking and also provides numerous tools and wizards to simplify programming the MC.

BASIC Moves also provides modern debugging features such as allowing task control by visually setting breakpoints, watch variables, and single stepping. BASIC Moves automatically displays the data recorded on the MC during operation. BASIC Moves provides all the tools you need for developing and debugging your application. When you start Basic Moves, your first action is to select a method for communicating with your MC.




If you are using either a PCI or ISA model MC, choose *ISA/PCI Bus*. If you are using a Stand-alone model MC, select either *Serial Port* or *Ethernet* (communication method configured for your system). For more information concerning communication with the stand-alone MC, refer to the Software Installation section of the *SERVO**STAR**® MC Installation Manual*.

## 2. 1 COMMUNICATION

Communicating with a stand-alone MC is not as automatic as is comunicating with a PCI or ISA plug-in MC. Assuming you have properly configured the communication method during installation of the BASIC Moves Development Studio on your host computer, there are still some operating procedures you may need to perform.

**Ethernet** If you configured Ethernet communications, subsequent communication with the MC 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. Refer to the *SERVO**STAR**® MC Installation Manual* for additional information.

**Serial** The installation package includes the Virtual NIC device driver, which is started automatically by Basic Moves Development Studio. No special configuration is required. Refer to the *SERVO**STAR**® MC Installation Manual* for additional information.

## 2. 2 MC-BASIC

The MC is programmed in MC-BASIC® (**M**otion **C**ontrol BASIC), a version of the BASIC programming language enhanced for multi-tasking motion control. If you are familiar with BASIC, you already know much of MC-BASIC. For detailed information on any of the commands (including examples) used in MC-BASIC, refer to the *SERVO**STAR**® MC Reference Manual*.

To develop your application on the MC, you must install BASIC Moves Development Studio. Refer to the software installation section of the *SERVO**STAR**® MC Installation Manual* for detailed instructions.

## 2.2.1. Instructions

Instructions are the building blocks of BASIC. Instructions set variables, call functions, control program flow, and start processes such as events and motion. In this manual we will use the terms ***instruction*** and ***command*** interchangeably. For detailed information on any of the instructions (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

**Syntax** is the set of rules that must be observed to construct a legal command (that is, a command the MC can recognize).

MC-BASIC is **line-oriented**. The end of the line indicates the end of the instruction. Whitespace (*i. e.*, 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 length of a line is 80 characters.

MC-BASIC is **case insensitive**. Commands, variable names, filenames, and task names may be written using either upper case or lower case letters. The only exception is that when printing strings with the "*Print*" and "*PrintUsing*" commands, upper and lower case characters can be specified.

**Syntax** uses the following **notation**:

[ ] 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
```

## 2.2.2. Type

There are many types of instructions: comments, assignments, memory allocation, flow control, task control, and motion. For detailed information on any of the commands (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

**Comments** allow you to document your program. Indicate a comment with either the Rem command or a single apostrophe ('). You can add comments to the end of an instruction with either Rem or an apostrophe.

```
Rem This is a comment
' This is a comment too.
X = 1 'This comment is added to the line X = 1
X = 1 REM This is a comment too
```

Use comments generously throughout your program. They are an asset when you need support from others and they avoid confusing code.

**Declarations** allocate MC memory for variables and system elements (groups, cam tables, etc.). This might be a simple type as an integer, or a complex structure as a cam table.

**Assignments** Assignment instructions assign a new value to a variable.

The syntax of an assignment is
[Lvalue] [=] [expression]

For example:

```
X = Y + 1
```

The term, *Lvalue*, is a shorthand notation, which indicates the value to the left of the equals sign. Valid *Lvalues* are variables or writable properties, which can be assigned. Expressions can be variables, constants, properties and function calls, as well as various combinations of them in arithmetic and logical statements . An exception to this rule is generic elements' assignment, at which the right side of the equal sign is not an expression, but an axis or a group (either real or generic), and Lvalue is a generic elements. If you assign a Double (floating point) value (expression, variable or constant) to a Long variable, the fractional portion of the double value is truncated, and the integer portion is assigned to the long variable. To query a variable or expression from the BASIC Moves terminal window, use the **PRINT** or **?** command:

```
PRINT 1/100
? X1
```

MC-BASIC also provides the standard **PRINTUSING** (**PrintU**) command for formatted printing.

Commands for **flow control** change the way your program is executed. Without flow control, program execution is limited to processing the line immediately following the current command. Examples of flow control include **GOTO**, **FOR…NEXT**, and **IF…THEN**.

MC-BASIC is a multi-tasking language in which many tasks can run concurrently. Generally, tasks run independently of each other. However, tasks can control each other using **inter-task control** instructions. One task can start, idle, or terminate another task.

Most commands are started and finished immediately. For example:

```
x = 1                ' this line is executed completely…
y = 2                ' …before this line is started
```

For general programming, one command is usually finished before the next command starts. Effects of these commands do not persist beyond the time required to execute them. However, the effects of many other commands persist long after the execution of the command. In general programming, for example, the effects of opening a file or allocating memory persist indefinitely. In real-time systems, **persistence** is more complicated because the duration of persistence is less predictable.

Consider a command that specifies a 1,000,000 counts move on an axis named, A2:

```
Move A2 1000000.0
Y = 2
```

The MC does not wait for the 1,000,000 move to be complete before executing Y=2. Instead, the first command starts the motion and then the MC continues with the next line (Y = 2). The move continues well after the move command has been executed.

Persistence can affect programs. For example, you may not want to start one move until the previous move is complete. In general, you need access to persistent processes if you are to control your program according to their state of execution. As you will see later, MC-BASIC provides this access.

# 2.2.3. Constants and Variables

All constant, variable and system element names 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. For detailed information on any of the constants and variables (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

## *2.2.3.1. CONSTANTS*

Constants are numbers, which are written as ordinary text characters; for example, 161. Constants can be written in decimal or in hexadecimal (HEX). To write a constant in hex, precede it by a 0x; for example, 0xF is the hex representation of 15 decimal.

The MC provides numerous **literal constants** (reserved words with a fixed value). You can use these constants anywhere in your program to make your code more intuitive and easier to read. For example:

```
A1.Motion = ON
```

turns on the A1 property Motion. This is more intuitive to read than:

```
A1.Motion = 1
```

although the effect is identical.

### *2.2.3.1.1 Literal Constants Table*

ABORT = 4
ABOVE = 2
BELOW = 1
CAM = 2
CAPTURING = 16
CLEARMOTION or CMOT = 3
CLOSE = 0
CONTINUE or CONT = 1
ENDMOTION or EMOT = 3
ERRORLEVEL = 2
EXTERNAL = 1
FALL = 2
FALSE = 0
FAULTLEVEL = 1
FLIP = 2
GEAR = 1
GENERATORCOMPLETED or GCOM = 3
HALT = 0
HOMING = 10
IDN_CAPTURE1ENABLE = 405
IDN_CAPTURE1POSITIVELATCHED = 409
IDN_CAPTURE1NEGATIVELATCHED = 410
IMMEDIATE or IMMED = 1
INPOSITION or INPOS = 2
LEFTY = 1
MAXDOUBLE = 1.797693134862311e+308
MAXLONG = 2147483647
MINDOUBLE = -1.797693134862311e+308
MINLONG = -2147483648
MULTIPLE = 1
NEGATIVE = -1
NEXTMOTION or NMOT = 2
NOFLIP = 1
NOOVERLAP = 0
NOTELEVEL = 3
OFF = 0
ON = 1
ONPATH = 2
OPEN = 1
OVERLAP = 1
PI = 3.14159265359
POSITIVE = 1
RESTART = 1
RIGHTY = 2
RISE = 1
SILENTLEVEL = 0
SUPERIMMEDIATE or SIMM = 5
SYNC = 4
TASK_ERROR = 4
TASK_IGNORE = -1
TASK_INCORRECT = 9
TASK_INTERRUPTED = 0x200
TASK_KILLED = 10
TASK_LOCKED = 0x100
TASK_READY = 7
TASK_RUNNING = 1
TASK_STOPPED = 2
TRACK = 3
TRUE = 1

## *2.2.3.2. VARIABLES*

You must declare a variable in MC-BASIC before you can it. In the declaration, you define variable name, scope and 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-like variables and some MC-BASIC specific types, such as points (Joints and Locations), generic motion elements (Axes and Groups) and UEAs (user error assertions-Errors and Notes).

**DeleteVar** deletes a global variable. Since variable name can include wildcards, a single **DELETEVAR** can be used to delete more than one variable.

```
DeleteVar int1
DeleteVar int*   ' Deletes all variables starting with int
```

Current values of global variables (both scalar and arrays) can be stored in a Prg file, in assignment format, within an automatically executable **Program Continue…Terminate Program** block. Obligatory parameters of **SAVE** are the name of storage file, and type of stored variables (could be all types). Optional parameters are robot type (for point variables), variable name (which may include wildcards) and mode of writing to storage file (overwriting or appending). Variable types available for storage through **SAVE** are longs, doubles, strings, joints and locations, but not structures, user-defined exceptions nor generic motion elements.

```
Save File = "IntFile.Prg" Type = All VariableName = "int*"
' Save all variables starting with int in IntFile.Prg (overwrite file)
Save File = "Points.Prg" Type = Joint RobotType = XYZR Mode = Append
' Append all joint-type points with XYZR robot-type to Points.Prg file
```

**Scope** defines how widely a variable can be accessed. The broadest scope is global. A global variable 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. Task variables can be read from or written to anywhere within the task where it is defined, but not from outside the task. Local variables can only be used within their declaration block, i.e. program, subroutine or function blocks. The scope of a variable is implicitly defined when a variable is declared using the keywords: Common, Shared, and Dim.

**Global variables** can be defined within the system configuration task, Config.Prg, within a task files (Prg files), before the program block, within library files (Lib files), before the first subroutine or function block, or from the terminal window of BASIC Moves. To declare a variable of global scope use the Common Shared instruction.

**Task variables** are defined within a task or a library. To declare a variable of task scope use the Dim Shared instruction.

**NOTE**

***All Dim Shared commands must appear above the Program statement in task files. In library files, all Dim Shared commands must appear above the first block of subroutine or function. The values of variables declared with Dim Shared persist as long as the task is loaded, which is usually the entire time the unit is operational. This is commonly referred to as being static.***

**Local variables** are defined and used within a program, a subroutine, or a function. To declare a local variable, use the Dim instruction. The Dim command for local variables must be entered immediately below the Program, Sub, or Function statement. Local variables cannot be declared within event blocks, and events cannot use local variables declared within the program.

# 2.2.4. Data Types

MC-BASIC has two **numeric data types**: 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 |

MC-BASIC provides the **string data type** which consists of a string of ASCII-coded characters. MC-Basic strings are dynamic. Reallocation of memory for new strings is preformed through a simple assignment of the string variable, and there is no need to specify the length of the new string. A full complement of string functions is provided to create and modify strings.

| Type | Description | Range |
|---|---|---|
| String | ASCII character string (no string length limit) | 0 to 255 (ASCII code) |

MC-BASIC has two **point data types**: JOINT and LOCATION. A point variable is related to a robot type. Robot type examples: XY – two axes XY table, XYZ – three axes XYZ system, XYZR – three cartesian axes + roll, etc.

| Type | Description | Range |
|---|---|---|
| Joint | A set of 2-10 double precision floating point joint (motor) coordinates | ±1.79769313486223157 E+308 for each coordinate |
| Location | A set of 2-10 double precision floating point cartesian coordinates | ±1.79769313486223157 E+308 for each coordinate |

MC-Basic enables definition of **structure-like data types**, composed of a limited number of long, double, string and point (joint and/or location) scalar and array elements. Array elements can have only a single dimension. Name of structure type and composition of elements are defined by the user within configuration file (Config.Prg).

| Type | Description | Range |
|---|---|---|
| 0-100 longs | 32 bit signed integer | -2,147,483,648 (MinInteger) to 2,147,483,647 (MaxInteger) |
| 0-100 doubles | Double precision floating point (about 16 places of accuracy) | ±1.79769313486223157 E+308 |
| 0-100 strings | ASCII character string (no practical limit to string length) | 0 to 255 (ASCII code) |
| 0-100 points (joints and/or locations) | A set of 2-10 double precision floating point coordinates | ±1.79769313486223157 E+308 for each coordinate |
| 0-10 long arrays | 1-32,767 32 bit signed integers | -2,147,483,648 (MinInteger) to 2,147,483,647 (MaxInteger) |
| 0-10 double arrays | 1-32,767 double precision floating point elements | ±1.79769313486223157 E+308 |
| 0-4 string arrays | 1-32,767 ASCII character strings | 0 to 255 (ASCII code) |
| 1-2 point (joints and/or locations) arrays | 1-32,767 sets of 2-10 Double precision floating point coordinates | ±1.79769313486223157 E+308 for each coordinate |

MC-Basic has two **UEA** (user error assertion) **data types** (severities): Error and Note. Each UEA is defined with a string-type error message given by the user and a unique exception number, which might be determined either by the user or by the system. The SERVO**STAR**® MC handles UEAs similar to internal exceptions.

| Type | Description | Range |
|---|---|---|
| Error | An ASCII message string and a unique integer exception number | 20001-20499 for user determined exception number<br>20500-20999 for system determined exception number |
| Note | An ASCII message string and a unique integer exception number | 20001-20499 for user determined exception number<br>20500-20999 for system determined exception number |

MC-Basic provides **generic element data types** designed to serve as a reference to real axes and groups. Through a simple assignment statement, the generic element gets the element identifier (see below) of a real motion element, thus acquiring all its properties. Afterwards, all activities preformed on the generic element identically affect the real element. The referenced element can be changed, through recurring assignments, as many times as the user wishes.

| Type | Description | Range |
|---|---|---|
| Generic Axis | An integer element identifier | 0 (before first assignment)<br>1 to number of axes in system (up to 32) |
| Generic Group | An integer element identifier | 0 (before first assignment)<br>33 to number of groups in system (up to 64) |

Each real motion element gets a unique element identifier from the system during its declaration. Axes get successive element identifiers ranging from 1 to 32 according to axis number. Groups get successive identifiers ranging from 33 to 64 according to declaration order. Value of element identifier can be queried directly through **ELEMENTID**.

```
SYS.NUMBERAXES = 2
Common Shared G2 As Group AxNm = A1 AxNm = A2
Common Shared G1 As Group AxNm = A1 AxNm = A2

? A1.ELEMENTID
1
? A2.ELEMENTID
2
? G2.ELEMENTID
33
? G1.ELEMENTID
34
```

MC-BASIC uses integers to support **Boolean (logical)** operations. Logical variables have one of two values: true or false. MC-BASIC uses type Long to support logical expressions. The value 0 is equivalent to false. Usually, 1 is equivalent to 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.

**Arrays** can be 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.

# 2.2.4.1. STRUCTURES

A structure is a new data type used for storing a list of variables of different type in one variable.

## 2.2.4.1.1 Definition

Since a structure is a user-defined data type; it must first be defined. Structure type definition can be done only in the Config.prg file (before the **PROGRAM** block, at the declaration level), using the following syntax:

```
TYPE <structure_type_name>
<element_name>{[]} AS <element_type> {OF <robot_type>}
…
END TYPE
```

This block defines the name of the structure type and the names, types and array sizes of the various structure elements (fields). Data types supported as structure elements are: long, double, string, and points (JOINT and LOCATION). Array elements can only have a single dimension. The maximum structure block can include: 100 longs, 100 doubles, 100 strings, 100 points JOINTs and/or LOCATIONs), 10 long-type arrays, 10 double-type arrays, 4 string-type arrays and 2 point-type arrays JOINTs and/or LOCATIONs). The total size of the MC-Basic structure is limited. The maximum number of each type of element is fixed. Trying to define more structure elements than allowed results in a translation error. An array of structures may have up to 10 dimensions.

## 2.2.4.1.2 Declare

You must declare a structure before you can use it. If you want to declare a structure variable, you must first declare a structure data type and then use this data type structure to declare the variable. The structure data type definition can be done in the Config.prg file only with the following syntax (before the **PROGRAM** token, at the declaration level):

```
TYPE <variable_name>
<variable_name> as <type>
<variable_name> as <type> {<of> <robot_type>}
…
END TYPE
```

This block defines the new element names of the structure type. The different types supported for the structure are: long, double, string, and points. The maximum structure block is defined as:

100 longs
100 doubles
100 strings
100 points

## 2.2.4.1.3 Syntax

```
(COMMON SHARED|DIM {SHARED}) <variable_name> AS <variable_name>
```

or

```
(COMMON SHARED|DIM {SHARED}) <variable_name>[<index>] AS
<variable_name>(arrays of structure)
```

where the second variable name = the structure type

For example:

```
In config file ->
Type X
Type as Long
Length as Long
End Type

In application file ->
Dim shared s1 as X
Program
?s1->Type
End program
```

Do not define the size of the structure data type. The size of the structure is constant. If you declare more structure elements than allowed, a translation error is generated.

#### *2.2.4.1.4 Assignment*

The assignment of a structure uses the following syntax:

`<variable_name> = <expression> where <expression> is of structure type`

Whole structure assignment is allowed only if both structures are of the same structure type. For example:

```
Dim STa1 As Type_1
Dim STb1 As Type_1
Dim STa2 As Type_2
STa1 = STb1                -> Structure types match
STa2 = STa1                -> Error - structure type mismatch
```

#### *2.2.4.1.5 Elements*

A structure element is addressed through the structure's name and the arrow sign. For example:

`<structure_name>{[]…}-><element_name>{[]}`

Structure elements are handled almost like regular variables. Structure elements are printed, assigned, participate in mathematical operations, and string-type elements are concatenated to other strings. They can also serve as arguments of system functions (**SIN**, **UCASE$**, etc.), used in logic statements and as conditions of flow control statements and event definitions. For example:

```
Common Shared ST As STRUCT

PRINT ST->LongElm1
ST->LongElm2 = 21.2 * ST->LongArrElm[1]
? UCASE$("String Element Is: " + ST->StringElm)
IF ST->LongElm1 > 0 THEN
SELECT CASE ST->LongElm2
```

## 2.2.5. System Elements

There are several system elements: axes, groups, cam tables, PLSs, conveyers and compensation tables. These are not regular data types, but are more like user interfaces of the system's internal memory or internal function calls. You cannot handle them as a whole in expressions. Each system element has a set of properties, which is accessed through syntax resembling data elements of a structure, using the point character.

Despite the syntax, properties are not data elements, since a system element and its properties are not located on a continuous block of memory. All properties return a value, so they can be printed, combined in expressions and passed by value to functions and subroutines. Many properties are also writable and can be assigned like variables.

Unlike variables, properties do not have a fixed address in memory and cannot be passed by reference. Data types of properties are Long, Double, String, Joint and Location. For axis A1 and cam table Cam1, and a long type variable named Var1:

```
? A1.Enable               ' Query
A1.Enable = 1             ' Set (read-write property)
? A1.PositionCommand      ' Query
A1.PositionCommand = 0    ' Syntax Error (read-only property)

Var1 = Cam1.Cycle         ' Query
Cam1.Cycle = -1           ' Set (read-write property)
Var1 = Cam1.Inuse         ' Query
Cam1.Inuse = 1            ' Syntax Error (read-only property)
```

## 2.2.6. Units

Many variables have associated units. This include variables, which contain quantities of position, velocity, acceleration, and time. Some units are fixed. For example, time is always in milliseconds. MC-BASIC allows you to define units of position, velocity and acceleration independently. In the next chapter, we will discuss how to set up units.

## 2.2.7. Expressions

Expressions are combinations of operators and value-returning entities, such as variables, constants, function calls, which are all calculated into a value. In MC-Basic, expressions can also contain unique entities like element properties (see above) and system properties (see below). Operators (like + and -) specify how to combine the variables and constants. Expressions are used in almost all commands.

An expression has one of two **syntax** forms

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

An operand can be a constant (123.4), variable name (X), or another expression. Most operators are binary (they take two operands), but some are unary (taking only one operand). So, -5.5 is a valid expression. It has the unary minus (also called negation) and a single operand.

There are two **types** of expressions: algebraic and logical (or Boolean).

**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.

## 2.2.8. 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 I as Long
I = 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.

**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 are considered interchangeable.

Logical expressions are evaluated from left to right. They are evaluated only far enough to determine the result. So, in `A And B And C`, if A is 0 (false), the result is false without evaluating B or C.

**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) + 1)`, not `9 ((2 + 1) * 3)`. 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.

The tables that follow list different types of operators. The entries in the tables are listed in order of precedence: the higher in the table, the higher the operator precedence. Also, the tables themselves are arranged in order of precedence with respect to one another. Math operators have the highest precedence. So, all math operations are executed before any other type of operator is executed. Finally, any time you need to change the standard precedence, use parentheses. Parentheses have the highest precedence of all.

| Algebraic Operator | Symbol | Unary/Binary | Operand Type |
|---|---|---|---|
| Parentheses | ( ) | NA | double, long, string, joint, location |
| Exponentiation | ^ | Binary | double, long |
| Negation | - | Unary | double, long, joint, location |
| Unary Plus | + | Unary | double, long, joint, location |
| Multiplication | * | Binary | double, long, joint, location |
| Division | / | Binary | double, long, joint, location |
| Modulo | Mod | Binary | double, long |
| Addition | + | Binary | double, long, string, joint, location |
| Subtraction | - | Binary | double, long, joint, location |
| Compound | : | Binary | location |

| Relational Operator | Symbol | Binary/Unary | Operand Type |
|---|---|---|---|
| Parentheses | ( ) | Not applicable | double, long, string |
| Equality | = | Binary | double, long, string |
| Inequality | <> | Binary | double, long, string |
| Less than | < | Binary | double, long, string |
| Greater than | > | Binary | double, long, string |
| Less than or equal to | <= | Binary | double, long, string |
| Greater than or equal to | >= | Binary | double, long, string |

| Word-wise Logical Operator | Symbol | Unary/Binary | Operand Type |
|---|---|---|---|
| And | And | Binary | long |
| Or | Or | Binary | long |
| Exclusive or | Xor | Binary | long |
| Not | Not | Unary | long |

| Bit-Wise Logical Operator | Symbol | Unary/Binary | Operand Type |
|---|---|---|---|
| Bitwise Negation | BNot | Unary | long |
| Bitwise And | BAnd | Binary | long |
| Bitwise Or | BOr | Binary | long |
| Bitwise Xor | BXor | Binary | long |
| Shift Left | SHL | Binary | long |
| Shift Right | SHR | Binary | 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, the resultant value is negative (-2). The sign of the result is determined by the sign of the first operand, 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 section of the above example, exponentiation is performed first then negation is performed.

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

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

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

In the example above, 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 strings, performs concatenation of strings.

In points, "+" (addition) and "-" (subtraction) operators can be used between points of the same type and size, or between a point and a long or double type scalar. On the other hand, "*" (multiplication) and "/" (division) operators can only be operated between a point and double or long scalar, but not between two points.

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. 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 expressions differ from logical expressions in that there may be many results from a single operation. Bitwise operations are frequently used to mask I/O bits. For example, the standard SERVO**STAR** MC inputs can be operated on with BAnd (Bitwise 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
```

All bitwise operations produce 32-bit result from two 32-bit numbers. Bitwise operations are really 32 independent, bit-by-bit operations. For example:

15 *(Binary 1111)* BAnd 5 *(Binary 101)* is 5 *(Binary 101)*
BNot 15 is 0xFFFFFFF0.

## 2.2.9. Math Functions

MC-BASIC provides a large assortment of mathematical functions. All take either numeric type as input (Double and Long) and all but Int and Sgn produce Double results. Int and Sgn produce Long results. For detailed information on any of the math functions (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

| | |
|---|---|
| **Abs(x)** | Returns the absolute value of X. |
| **Acos(x)** | Returns the arc cosine of X. X is assumed to be in radians. |
| **Asin(x)** | Returns the arc sine of X. X is assumed to be in radians. |
| **Atan2(Y, X)** | Returns the inverse tangent of Y/X in radians. The range of the result is between ±PI radians (±180°). Atan2 is an improvement over ATN(Y/X) in that it works correctly if X is zero, and if X<0. |
| **Atn(X)** | Returns the inverse tangent of X in radians. The range of the result is between ± PI/2 radians (± 90°). |
| **Cos(X)** | Returns the cosine of X. X is assumed to be in radians. |
| **Exp(X)** | Returns ex. |
| **Int(X)** | Returns the largest long value less than or equal to a numeric expression (for example, ?Int(12.5) returns 12<br>?Int(-12.5) returns -13) |
| **Log(X)** | Returns the natural logarithm of X. X must be > 0. If you want the base-10 log, use Y = LOG(X)/LOG(10). |
| **Sgn(X)** | Returns the arithmetic sign of X. If X > 0, return 1. If X < 0, return -1. If X = 0, return 0. |
| **Sin(X)** | Returns the sine of X. X is assumed to be in radians. |
| **Sqrt(X)** | Returns the positive square root of X. X must be >= 0. |
| **Tan(X)** | Returns the tangent of X. X is assumed to be in radians. |
| **Round(X)** | Returns the integer nearest to X. If X falls exactly between two integers (for example, 1.5), return the even integer. So 1.5 and 2.5 round to 2. |

# 2.2.10. String Functions

Beginning with version 3.0, MC-BASIC supports all the common string functions supported in standard BASIC. For detailed information on any of the string functions (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

| | |
|---|---|
| ASC(S,I) | Returns an ASCII character value from within a string, S, at position, I. |
| BIN$(X) | Returns the string representation of a number (X) in binary format (without the Ob prefix). |
| CHR$(X) | Returns a one-character string corresponding to a given ASCII value, X. |
| HEX$(X) | Returns the string representation of a number (X) in hexadecimal format (without the 0x prefix). |
| INSTR(I,SS,S) | Returns the position, I, of the starting character of a substring, SS, in a string, S. |
| LCASE$(S) | Returns a copy of the string, S, passed to it with all the uppercase letters converted to lowercase. |
| LEFT$(S,X) | Returns the specified number, X, of characters from the left-hand side of the string, S. |
| LEN(S) | Returns the length of the string, S. |
| LTRIM$(S) | Returns the right-hand part of a string, S, after removing any blank spaces at the beginning. |
| MID$(S,I,X) | Returns the specified number of characters, X, from the string, S, starting at the character at position, I. |
| RIGHT$(S,X) | Returns the specified number of characters, X, from the right-hand side of the string, S. |
| RTRIM$(S) | Returns the left-hand part of a string, S, after removing any blank spaces at the end. |
| SPACE$(X) | Generates a string consisting of the specified number, X, of blank spaces. |
| STR$(X) | Returns the string representation of a number, X. |
| STRING$(X,{S},{Y}) | Creates a new string with the specified number, X, of characters, each character of which is the first character of the specified string argument, S, or the specified ASCII code, Y. |
| UCASE$(S) | Returns a copy of the string, S, passed to it with all the lowercase letters converted to uppercase. |
| VAL(S) | Returns the real value represented by the characters in the input string, S. |

# 2.2.11. System Commands

The following commands provide information about the system. You can issue these commands at any time either from task, terminal or the CONFIG.PRG file (with some exceptions). For detailed information on any of the system commands (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

| | |
|---|---|
| AxisList | Returns a comma-separated-list of the axes' names defined in the system. If wildcards are used, the query returns the proper axes. |
| BreakPointList | Lists all the breakpoints in the task, loaded to memory. |
| CamList | This query returns a list of the cam table names defined in the system. If wildcards are used, the query returns the proper existing cam names. |
| Dir | Lists all files that exist on the RAM disk and on the Flash disk. The File Specification may contain the * and ? wildcard characters, in order to refine the list. When invoked without a parameter, all the files are listed. |
| ErrorHistory | Displays the log file containing 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. If a task name precedes **EventList**, only events belonging to the specified task are listed. |
| GroupList | Lists the names of the groups defined in the system. Each group name is followed by a list of the axes' names that are part of that group. |
| PlsList | Returns a list of the PLS names defined in the system. If wildcards are used, the query returns the correct existing PLS data. |
| ProgramPassword | Sets the file password and toggles the password protection state. |
| Reset All \| Tasks | **Reset All** removes all tasks, variables, and system variables from program memory. All external outputs are turned off and all drives are disabled. **Reset All** runs Config.prg, not Autoexec.prg. **Reset All** is only issued from the terminal.<br><br>**Reset Tasks** removes all tasks from memory, but leaves system variables loaded. It deallocates all defined variables and user programs. **Reset Tasks** does not disable motors, or reset outputs. **Reset Tasks** is only issued from the terminal. |
| System.AccelerationRate | Queries or sets the system acceleration exchange rate. |
| System.AutostartTask | Queries or sets the name of the auto-start task. The default auto-start task is Autoexec.Prg. Assignment of an empty string prevents auto-start task run. |
| System.AverageLoad | Returns the average realtime load on the CPU. This is the average realtime load measured during a 0.5 second interval. |
| System.Clock | Returns the number of system clock ticks. This is the clock run by the VxWorks Operating System. One clock tick corresponds to 1 millisecond. |
| System.CpuType | Returns the type and model of the CPU that was found during the system's power-up. Supports only Intel, AMD and Cyrix processors. |
| System.Date | Queries or sets the date. The date is entered as a character string and must be enclosed between double quotes ("). The parameters Day, Month, and Year, must be separated with the / (slash) character. |

| | |
|---|---|
| System.DecelerationRate | Queries or sets the system deceleration exchange rate. |
| System.DIn | Returns the value of one or more of the 23 digital inputs. **System.DIn** returns the values of the individual bits in the system input word. In order to read the value of a single input bit, bit number should be specified as **System.DIn**.<*bit number*>. |
| System.DipSwitch | Returns the value of one or more of the eight DIPswitch inputs. **System.DipSwitch** returns the values of the eight individual bits in the DIPswitch byte. In order to read the value of a single input bit, bit number should be specified as **System.DipSwitch**.<*bit number*>. |
| System.DiskFreeSpace | Returns the amount of free space on the flash disk. |
| System.DoubleFormat | Queries and sets the printing format of Doubles. Zero value stands for d.ddddddddddddddde+/-dd style, with precision of 15 digits after the decimal point. 1 relates to the ddd.ddd style if the exponent is between -4 and 5, and to d.de+/-dd style otherwise. Setting can be done only in Config.Prg file, and therefore requires system reset (through **RESET All** or hardware reset). |
| System.DOut | Queries and sets the value of one or more of the 20 digital outputs. **System.DOut** sets or returns the values of the individual bits in the system output word. In order to read or write into a single output bit, bit number should be specified as **System.DOut**.<*bit number*>. |
| System.Enable | Enables and disables the system. |
| System.Error | Queries the last system error message. |
| System.ErrorHandlerMode | Determines the scope of the effect of an error occurring in task context. When the value of **System.ErrorHandlerMode** is one (the default value), the error will affect the entire system by stopping all motion, idling all tasks that have attached elements and turning off System.Motion flag. Setting this property to zero results in limiting the Motion stop and task idling to the erroneous task itself. |
| System.ErrorNumber | Queries the number of the last system error and returns only the error number. |
| System.ErrorPrintLevel | Controls which errors are printed, according to the severity of the error. Only asynchronous errors are printed. Synchronous errors are not affected. |
| System.FlashDiskSize | Returns the size of the Flash disk. |
| System.HostDouble | Reads the value of one of eight double-type DPRAM variables written by the host. |
| System.HostInteger | Reads the value of one of eight long-type DPRAM variables written by the host. |
| System.Information | Returns information found during system power-up. Performs a test on the SERCON chip cycle time. The result is the average cycle time found in a 500 ms test. |
| System.IPAddressMask | Queries or sets the IP address and subnet mask for the Ethernet interface. |
| System.JerkRate | Queries or sets the system jerk exchange rate. |
| System.Led | Queries and sets one or more of the three general purpose LEDs. In order to read or write into a single LED, LED number should be specified as **System.LED**.<*LED number*>. |
| System.MaxMemBlock | Returns the size (in bytes), of the largest block of free memory. |
| System.MCDouble | Reads or writes to one of eight double-type DPRAM variables available for use by the MC. |
| System.MCInteger | Reads or writes to one of the long-type DPRAM variables available for use by the MC. |
| System.Motion | Enables and disables Motion commands. |

| | |
|---|---|
| System.MotionAssistance | Enables and disables display of additional notes returned from the motion task. |
| System.Name | Queries and sets the name of the controller. |
| System.NoMotion | Enables and disables the **System.Motion** monitoring apparatus. |
| System.NumberAxes | Returns the number of axes currently loaded in the system. This is a read-only variable. It can be used in the terminal window or in any task. |
| System.PeakLoad | Returns the peak realtime load on the CPU. This is the maximum value of the realtime load measured during a 0.5 second interval. |
| System.PipeMode | Queries and sets the type of motion pipe mode. Available types are: active, position only, or position and velocity. |
| System.PrintMode | Queries and sets the printing format of Longs. Optional formats are DECIMAL (decimal format), HEX (hexadecimal format) and BIN (binary format). Default format is decimal. |
| System.RamDriveFreeSpace | Returns the amount of free space on the RAM drive. |
| System.RamSize | Returns the size of the RAM found during the system power-up. |
| System.RtsTimeout | Queries and sets the timeout of the real-time scheduler. |
| System.SerconVersion | Returns the version number of the SERCON chip. |
| System.SerialNumber | Returns the serial number of the MC. |
| System.ServicePrintLevel | Controls (query or set) printing of service messages (ON or OFF). |
| System.Time | Queries or sets the current time of day. Hours, minutes, and seconds are each specified using two digits. The MC uses 24-hour clock notation. Time is entered as a character string and must be enclosed between double quotes ("). The parameters, Hours, Minutes, and Seconds, must be separated with : (colon) characters. |
| System.UserAuthorizationCode | Returns the user authorization code. |
| System.VelocityOverride | Queries or sets the system velocity override. |
| System.VelocityRate | Queries or sets the system velocity exchange rate. |
| System.VIn | Returns the value of one or more of the 32 virtual inputs. **System.VIn** returns the values of the individual bits in the system input word. In order to read the value of a single input bit, bit number should be specified as **System.VIn**.<*bit number*>. |
| System.VOut | Queries and sets the value of one or more of the 32 virtual outputs. **System.VOut** sets or returns the values of the individual bits in the system output word. In order to read or write into a single output bit, bit number should be specified as **System.VOut**.<*bit number*>. |
| System.WatchDogTest | Checks the correct operation and timeout of the WatchDog. After the test has finished, the MC must be rebooted. The test returns either the WD timeout (in ms) when executed successfully, or an error in case of a test failure. |
| TaskList | Lists the names and states of loaded tasks. |
| VarList | Returns a list of the variable names defined in the system. |
| Version | Returns information pertaining to the version of BASIC Moves Development Studio loaded into your MC. |

With…End With

Simplifies blocks of code, which set a number of parameters on the same motion element (axis or group). After a **With** is encountered, all parameters where the element is not specified are assumed to belong to the element specified in the **With**. **With** is valid in Configuration, Task, or Terminal contexts. The scope of a configuration **With** is global, whereas the scopes of both terminal and task **With**'s are limited to terminal and task, respectively.

To put off the **With**'s effect, use **End With**. A task **With** must be followed by the **End With**, thus creating a closed **With** block. A **With** block must be closed within the same block it was opened (i.e., Program, Sub and Function blocks). For example:

```
A1.VMax = 5000
A1.Vord = 5000
A1.VCruise = 3000
A1.PEMax = 10
A1.PESettle = 0.01

Move A1 100
```

Can be simplified using **With**:

```
With A1

VMax = 5000
     Vord = 5000
     VCruise = 3000
     PEMax = 10
     PESettle = 0.01

     Move 100
End With
```

## 2.2.12. Printing

MC-BASIC provides unformatted and formatted printing using the PRINT and PRINTUSING commands. For detailed information on any of the print commands (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

The **Print** command (short form, **?**) is used to print expressions from the terminal window or from your program. If you issue print commands from the terminal window, they print to the terminal window. If you issue them from your program, they print to the BASIC Moves Message Log. You can view the message log by selecting Window, Message Log.

Within programs, **PRINT** allows you to suppress the carriage return at the end of a line by terminating the command with a comma or a semicolon. Semicolon and comma have no effect when added to the end of print commands from the terminal window.

```
Print "Hello, ";
Print " world."                'Prints:
  Hello, world.
```

When commas are added to print command as separators between expressions, a tabular space is placed between the expressions in the printed outcome.

```
Print "Hello, ",
Print " world."              'Prints:
  Hello,  world.
```

On the other hand, expressions separated by a semicolon or a space in the print command are printed adjacently.

```
Print 1,2
1   2

Print 1;2
12

Print 1 2
12
```

**PrintUsing** introduces formatting for printing. You can control the format and the number of digits that print. For example, by using the "#" sign and the decimal point, you can specify a minimum number of characters to be printed.

```
PrintU "The number is #, # ";j1,j2
-->The number is 1, 2
```

Be aware that using a single format to print more than one expression will result in repetition of any text written within string format according to the number of printed expressions.

```
PrintU "The number is #, ";j1,j2
-->The number is 1, The number is 2,
```

By specifying the number of digits to print, data can be printed in tabular form since the number of places printed will not vary with the value of the number.

```
PrintU "Keep numbers to a fixed length with PrintUsing:
######" ; 100
```

Be aware that if the number requires more digits than you have allocated, it overruns the space.

```
PrintU "Overrun: ##" ; 1000000
```

Takes 7 spaces for 1,000,000 even though **PrintU** allocates only 2.

For printing of double type expressions, formatted printing also enables to control precision, i.e. number of digits printed after the decimal point.

```
PrintU "Print PI with 3 digits after decimal point: #.###" ; 3.14159
```

Prints only 3 digits after the decimal point, while rounding the PI's value to 3.142.

You can also precede the "#" signs with a "+" to force PrintU to output a sign character (+ or -). Normally, the sign is printed for negative numbers, not positive numbers.

Finally, you can terminate the format string with "^^^^" to force the number to print in exponential format.

```
PrintU "Exponential format: ####.##^^^^" ; 100
```

Prints 1e+02. You must include exactly four "^" characters for this format to work.

Be aware that while **PRINT** and **PRINTUSING** allow you to print out many expressions on a single line, these expressions are not **synchronized** to each other. The values can and usually do come from different servo cycles. If you are monitoring motion and need the command to be synchronized, you must use **Record**.

Normally, printing of longs is done in decimal format. To change printing to **Hexadecimal** format, type:

```
System.PrintMode = HEX
```

To change printing to **Binary** format, type:

```
System.PrintMode = BIN
```

To restore printing to **Decimal** format, enter:

```
System.PrintMode = DECIMAL
```

Printing double floating point numbers is not affected by **System.PrintMode**. As an alternative to **PrintMode**, you can place the keywords Decimal, Hex or Binat the end of **Print** or **PrintUsing** to print expressions in these formats. Subsequent print instructions (**?**, **PRINT**, **PRINTU**, and **PRINTUSING**) are not affected.

**INPB**, **INPW**, **PEEKB**, and **PEEKW** return long values. If you query any of these functions while in PrintMode decimal, the result is misleading. Change to Hex or Bin for the correct result.

**System.DoubleFormat ,** defined from the configuration file , sets the double number printing format for print of double-type numbers.
System.DoubleFormat = 1 – print in Floating point format ( 132.555 )
System.DoubleFormat = 0 - print in Exponential format ( 1.3255500000000000e+02 )

```
Program
   Sys.DoubleFormat = 1
End Program
-->?Sys.DoubleFormat
-->1
```

## 2.2.13. Flow Control

Flow control is the group of instructions that control the sequence of execution of all other instructions. For detailed information on any of the flow control commands (including examples), refer to the *SERVO**STAR**® MC Reference Manual*.

**If…Then** is the most basic flow control statement. The syntax of If…Then is:

If *condition* Then
*<first statement to execute if condition is true>*
{*multiple statements to execute if condition is true*}
{Else
*<first statement to execute if condition is false>*
{*multiple statements to execute if condition is false*}}
End If

where:

**If…Then** must be followed by at least one statement.

Else is optional, but if present must be followed by at least one statement.

***There is no Else if***. If you use an If after an Else, you must place the If on a new line.

**Select…Case** is an extension of **If…Then**. Anything that can be done with **Select…Case** can also be done with **If…Then**. Many times, **Select…Case** simplifies programming logic.

On the first line of a Case block of commands, you specify the variable or expression you want tested. For example:

```
Select Case I
```

Tests the variable I for a range of conditions.

You can also select expressions:

```
Select Case I - M/N
```

After you have specified the variable or expression, list one or more values or value ranges that the variable can take. There are four ways you can specify cases:

Exact Value
Logical Condition
Range
Else

The syntax of Select…Case is:

Select Case *SelectExpression*
{Case *Expression1*
{*statements to be executed if SelectExpression = Expression1*}}
{Case *Expression2*
{*statements to be executed if SelectExpression = Expression2*}}
{Case Is RelationalOperator *Expression3*
{*statements to be executed if the condition is true*}}
{Case *Expression4* To *Expression5*
{*statements to be executed if SelectExpression is between values*}}
{Case Else
{*statements to be executed if none of the above conditions are met*}}
End Select

where

SelectExpression is a Long, Double or String expression in Case…To…, if Expression4 > Expression5, the case is never true; no error is flagged.

Select…Case block. The following example puts all four types of cases together:

```
Program
Dim N as Long
   Select Case N
      Case 0
       Print "N = 0"
       Case 1
       Print "N = 1"
       Case is >=10
       Print "N >= 10"
       Case is < 0          'No requirement for statements after Case
       Case 5 TO 9
          Print "N is between 5 and 9"
       Case Else
       Print "N is 2, 3, or 4"
   End Select
End Program
```

**For…Next** statements allow you to define loops in your program. The syntax is:

For *counter* = *Start* To *End* {Step *Size*}]
{*Loop Statements*}
Next {*counter*}

where

If *Size* is not defined, it defaults to 1.
The loop is complete when the *counter* value exceeds *End*. For positive *Size*, this occurs when *counter>End*. For negative *Size*, when *counter<End*.
*Counter*, *Start*, *End*, and *Size* may be Long or Double.

For example:

```
For I = 2 TO 5
     Print "I = " I          'Prints 2, 3, 4, 5
Next I

For I = 4 TO 2 STEP –0.5
     Print "I = " I          'Prints 4.0, 3.5, 3.0, 2.5, 2.0
Next
```

**While…End While** allows looping dependent on a dynamic condition. For example, you may want to remain in a loop until the velocity exceeds a certain value.
The syntax of **While** is:

While *Condition*
{*statements to execute as long as condition is true*}
End While

where

The condition is evaluated before any statements are executed. If the condition is initially false, no statements are executed.

Statements are optional. If none are included, **While…End While** acts as a delay.

You can have any number of statements (including zero) to be executed.

For example:

```
Program
     While A2.VelocityFeedback < 1000
          Print "Axis 2 Velocity Feedback still under 1000"
     End While
End Program
```

Using the **While** or **Do…Loop**, the CPU repeatedly executes the **While** block (even if the block is empty). This is sometimes a problem. These commands do not relinquish system resources to other tasks. If you want to free up CPU resources during a **While** block or **Do…Loop**, include the **Sleep** command within the block as follows:

```
Program
     While A1.VCmd < 1000
          Sleep 1
     End While
End Program
```

The **Do...Loop** is similar to **While** except that the statement block is executed before the first evaluation of the condition. With the **Do...Loop**, the statement block are always executed at least once. **Do...Loop** also allows an option on the condition. Use **While** to execute while the condition is true, or **Until** to execute while the condition is false. The syntax of the **Do...Loop** is:

Do
{*statements*}
LOOP *While*|*Until Condition*

where:

The statements are executed at least once.

Statements are optional. If none are included, **Do...Loop** acts as a delay.

For example:

```
Dim Shared i as Long
Program
i = 0
Do
     i = i + 1
     Print i
Loop Until i = 10
End Program
```

or, equivalently, you can use **Loop While**:

```
Dim Shared i as Long
Program
i = 0
Do
     i = i + 1
     Print i
Loop While i < 10
End Program
```

**GoTo** unconditionally redirects program execution. The syntax is:

GoTo *Label1*
…
*Label1*:

where:

*Label1* is a valid label within the same task as the **GoTo**.

The label must be on a separate line.

You can only branch within a Program, Event, Function, or Subroutine.

You can **GoTo** a label placed within a program block (**If…Then**, **For…Next**, **Select…Case**, **While…End While**, and **Do...Loop**) only if the **GoTo** and label are within the same block.

If the program is within a program block, you cannot **GoTo** a label outside that block.

Avoid **GoTo** wherever possible. History has shown that excessive use of **GoTo** makes programs harder to understand and debug. Use the other program control statements whenever possible.

## 2.2.13.1. ERROR TRAPPING

There are four ways to specify catch statements: Exact Value, Logical Condition, Range, and Else. The syntax used is:

Catch *Error_Number*
{*statements to execute if error Error_Number had occurred*}
Catch Is <*RelationalOperator*> *Error_Number*
{*statements to execute if the condition is true*}
Catch *Error_Number1* To *Error_Number2*
{*statements to execute if error number is between values*}
Catch Else
{*statements to execute if all other errors had occurred*}

where:

The number of **Catch** statements is not explicitly limited.

*Error_Number*s can only be long-type numeric values.

In **Catch…To…**, if *Error_Number1* > *Error_Number2,* the catch statement is never true and no error is flagged.

**Catch** statements are used within three types of error trapping blocks.

The **Try** block is designed to trap synchronous errors within task context (For more details, see the *Error Handling* section). There is no explicit limitation on the number of Try blocks instances in a program. The syntax for a Try block is:

Try
{*code being examined for synchronous errors*}
{Catch *Error_Number*
{*statements to be executed*}}
{Catch Is <*RelationalOperator*> *Error_Number*
{*statements to be executed*}}
{Catch *Error_Number1* To *Error_Number2*
{*statements to be executed*}}
{Catch Else
{*statements to be executed*}}
{Finally
{*statements to be executed if an error was trapped*}}
End Try

An example of a Try block, designed to catch errors in the loading process of Task1.Prg:

```
Try
     Load Task1.Prg
Catch 4033                    ' File does not exist
     Print "Task not found"
Catch 6007                    ' Task must be killed first
     KillTask Task1.Prg
     Unload Task1.Prg
     Load Task1.Prg
Catch Else
     Print "Error while loading Task1.Prg"
Finally
     Print "Caught error: " Task1.prg.Error
End Try
```

The **OnError** block is designed to trap and process both synchronous and asynchronous errors in a task. OnError traps errors not trapped by the Try/Finally mechanism within the task. (For more details, see "Error Handling" section). Only one instance of OnError may exist in a program. The syntax for OnError block is:

OnError
{Catch *Error_Number*
{*statements to be executed*}}
{Catch Is <*RelationalOperator*> *Error_Number*
{*statements to be executed*}}
{Catch *Error_Number1* To *Error_Number2*
{*statements to be executed*}}
{Catch Else
{*statements to be executed*}}
End OnError

An example of an **OnError** block, designed to stop motion in case of a motion error:

```
OnError
Catch 3001 To 3999                ‘ Motion errors
   System.Motion = 0
   A1.Enable = 0
   ? VESExecute("System.Motion = 1")
   A1. Enable = 1
   Print "Caught a Motion error: " ThisTask.Prg.Error
Catch Else
Print "Caught a non-Motion error: " ThisTask.Prg.Error
End Onerror
```

The **OnSystemError** block is designed to trap and process both synchronous and asynchronous errors in all tasks, as well as errors that occur within the context of the system (For more details, see the *Error Handling* section). Only one instance of **OnSystemError** may exist in the system. The syntax for **OnSystemError** block is:

OnSystemError
{Catch *Error_Number*
    {*statements to be executed*}}
{Catch Is <*RelationalOperator*> *Error_Number*
    {*statements to be executed*}}
{Catch *Error_Number1* To *Error_Number2*
    {*statements to be executed*}}
{Catch Else
    {*statements to be executed*}}
End OnSystemError

An example of an **OnSystemError** block, designed to monitor errors in task Errors.Prg:

```
OnSystemError
Catch Is < 12000
   Print “Caught a MC error: ” System.Error
   ‘ MC errors
   KillTask Errors.Prg
Catch Is > 20000
   Print “Caught a user error: ” System.Error
  ‘ User defined errors
   KillTask Errors.Prg
End OnSystemError
```

## 2.2.13.2. NESTING

Program control commands can be nested. Nesting is when one program control command (or block of commands) is within another. There is no specified limit on the number of levels of nesting.

For example, the following program nests a **WHILE…END WHILE** sequence within a **FOR…NEXT** sequence:

```
For I = 1 to 10
   N=5
   While N>0
      N=N-1      'This command will be executed 50 times
   End While
Next I
```

There is no specified limit on the number of levels of nesting. The Sample Nesting Program in Appendix A shows numerous combinations of nesting.

# 2. 3 PROGRAM DECLARATIONS

You must declare the start of programs and subroutines. For programs, use **Program…End Program**. Use **Sub…End Sub** keywords for subroutines.

## 2.3.1. Program

The **Program…End Program** keywords mark the boundary between the variable declaration section and the main program. Each task must have only one **Program** keyword, which ends with the **End Program** keyword. Another option is the **Program Continue…Terminate Program** block. In this case, the program is automatically executed after loading, and automatically unloaded from memory when it ends.

```
{Import of libraries}
{Declaration of global and static variables}

PROGRAM
…
END PROGRAM

{Local functions and subroutines}
```

## 2.3.2. Subroutine

Parameters (either scalar or array) passed to the subroutine are used within the code of the subroutine. The declaration line for the subroutine (**SUB**<***name***>) is used to declare names and types of the parameters to pass.

Parameters are passed either by reference or by value (ByVal). The default method is to pass parameters by reference. Whole arrays are passed only by reference. Trying to pass a whole array by value results in a translation error. However, array elements can be passed both by reference and by value. The syntax for a subroutine is:

SUB <*name*> ({<*par_1*>([*])+ as <*type_1*>}…{, <*par*_n>([*])+ as<*type*_n>})
…
END SUB

<*par_l*>: name of array variable

<*par*_n>: name of array variable

[*]: dimension of an array without specifying the bounds

+: means one or more [*]

```
SUB CalculateMean(x[*][*] as DOUBLE, TheMean[*] as LONG)
DIM sum as DOUBLE
DIM I as LONG
   FOR i = 1 to 100
      sum = sum + x[i][1]
   NEXT i
   TheMean[1] = sum/100
END SUB

SUB PrintMean(ByVal Mean as LONG)
   PRINT "Mean Value Is ", Mean
END SUB

CALL CalculateMean(XArray, TheMeanArray) ' Pass entire array by reference

CALL PrintMean(TheMeanArray[1])    ' Pass a single array element by value
```

When a variable is passed by reference (whether the variable is local to the task or global), the address of the variable is passed to the subroutine, which changes the value of the original variable (if the code of the subroutine is written to do this).

When a variable is passed by value (ByVal) a local copy of the value of the variable is passed to the subroutine, and the subroutine cannot change the value of the original variable.

There is no explicit limit on the number of subroutines allowed in a task. All subroutines must be located following the main program and must be contained wholly outside the main program. Subroutines can only be called from within the task where they reside. Subroutines may be recursive (can call itself). Use **CALL** to execute a subroutine with the following syntax:

**CALL** <subroutine_name>{(<par_1>{, …<par_n>})}

Parentheses are not used in subroutine **CALL** if no parameters are passed.

MC-BASIC automatically checks the type of compliance between the subroutine declaration and the subroutine call. Any mismatch (in number of parameters or parameters types) causes an error during program loading. Automatic type casting applies only for “by value” long and double parameters.

```
SUB MySub(RefPar as long, byval ValPar as double)
…
END SUB

CALL MySub(LongVar, “String”) -> type mismatch in second parameter
CALL MySub(LongVar) -> wrong number of parameters
CALL MySub(LongVar, 2) -> a valid type casting for a by-value parameter
CALL MySub(DoubleVar, 2.2) -> invalid type casting for a by-ref parameter
```

## 2.3.3. User-Defined Functions

MC BASIC allows the definition of user functions to be used in programs in the same manner as using BASIC's pre-defined functions. User-defined functions are composed with multiple lines and may be recursive (can call 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, function calls should be treated as expressions. Therefore, function called can be combined within print commands, assignment statements, mathematical operations and conditions of flow control statements. They can also be passed as by-value parameters of system or user-defined functions and subroutines.

**PRINT** <*function_name*>{(<*par_1*>{, …<*par*_n>})}
  <*variable_name*> = <*function_name*>{(<*par_1*>{, …<*par*_n>})}
**IF** <*function_name*>{(<*par_1*>{, …<*par*_n>})} > 10 THEN
  ? **LOG(** <*function_name*>{(<*par_1*>{, …<*par*_n>})} )

Parentheses are not used in function **CALL** if no parameters are passed. Parameters (either scalar or array) passed to the function are used within the code of the function. Declare variable names and types of parameters in the declaration line of the function. Parameters can be passed by reference or by value. The default is by reference.

Arrays can only be passed by reference. Trying to pass a whole array by value results in a translation error. On the other hand, array elements can be passed by reference and by value.

To set up the return value, assign the return value to a virtual local variable with the same name as the function somewhere within the code of the function. This local variable is declared automatically during function declaration and the function uses it to obtain the return value.

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 of the main program.

MC-BASIC automatically checks the type of compliance between the function declaration and the function call. Any mismatch (in number of parameters, in parameters and returned value types) causes an error during program loading. Automatic type casting applies only for long and double returned values and "by value" parameters. For example:

```
Function LongReturnFunc(…) As Long
   LongReturnFunc = "String"       -> type mismatch in returned value
End Function

Function LongReturnFunc(…) As Long
   LongReturnFunc = 43.7           -> valid type casting in returned value
End Function
```

A function can be recursive (can call itself). The following example defines a recursive function to calculate the value of *N*:

```
FUNCTION Factorial (ByVal N As Long) As Double
'By declaring N to be 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 a recursive function, you must have an **IF** statement to force the function to return without the recursive call being executed. Otherwise, the function never returns once it is called.

## 2. 4 LIBRARIES

Some applications repeatedly use the same subroutines and functions. Such subroutines and functions can be gathered into a library file. The library file can be imported to another task with **IMPORT** on the **.**lib file at the beginning of the task. Then, each public subroutine and function defined in the imported library file can be called within the importing task.

An MC-BASIC library is an ASCII file containing only the subroutines' and functions' code. The file does not have a main program part. The name of a library file must have the extension, **.LIB**. The format of a library file is:

```
Declarations of static variables

{PUBLIC} SUB <sub_name>
{declaration of variables local to the sub-program}
{sub code}
END SUB
…
{PUBLIC} FUNCTION <function_name> AS <return_type>
{declaration of variables local to the function-program}
{function code}
END FUNCTION
…
```

The **Public** keyword allows a subroutine or a function to be called from within the importing task. These subroutines and functions are visible from outside the scope of the library. Subroutines and functions declared without the **Public** keyword can only be used inside the scope of the library. They are PRIVATE library functions. For example:

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

SUB private_sub(…)
…
END SUB
```

The subroutine, private_sub, can be used only inside the library (i.e., by other subroutines and functions of the library). public_sub can be used in any user task (program file, another library file) importing the library. However, the same library can be imported by multiple tasks simultaneously.

A library file is handled like any other program file. A library file can be sent to the Flash disk, or retrieved and deleted from Flash disk. To use a library file in a program, the library must first be loaded into memory from either the terminal or another task.

```
LOAD MyLib.lib
```

To unload the library, enter:

```
UNLOAD MyLib.lib
```

If a program or library task references a subroutine or a function of a library, the program or 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. **IMPORT** must be placed at the beginning of the program code before any other command or declaration.

```
IMPORT MyLib.lib
```

The syntax for calling a library subroutine or function from the importing task is the same as the syntax for calling a local subroutine or function. There is no need to specify which library file contains the called subroutine or function (because a task cannot import two libraries defining subroutines or functions with identical names).

### 2.4.1. Global Libraries

Global libraries are library files (.LIB), which, instead of being loaded from either the terminal or another task, are loaded from the configuration file (Config.Prg). Another option is loading from terminal, using **LOADGLOBAL**.

PUBLIC subroutines and functions defined in such libraries can be called from everywhere (i.e., terminal and other tasks), without first being imported by the calling task. **IMPORT** cannot be applied to global libraries.

```
LOAD GlobLib.lib          ' In Config.Prg
LOADGLOBAL GlobLib.lib      ' In terminal

IMPORT GlobLib.lib → Error   ' In task or library
```

The syntax for calling a global library's subroutine or function from either the terminal or task is the same as the syntax for calling a subroutine or function of a regular (imported) library. Because of the wide recognition scope of global, public functions and subroutines, the names of such subroutines and functions cannot be used for another function or subroutine, variable, motion element, etc. As in regular libraries, subroutines and functions declared without the **PUBLIC** keyword can only be used inside the scope of the library.

A global library cannot be unloaded if there are task files (.PRG) loaded in memory. To unload a global library, all .Prg files must first be unloaded. If a global library file is unloaded and then reloaded from either the terminal or a task, it becomes a regular library, which must be imported before use.

## *2. 5 C-FUNCTIONS*

The MC offers users an outstanding option to incorporate applications written in C or C++ into an MC-Basic task.You can write your algorithm in C/C++, compile it with a GNU compiler, download it into a target program and call the code from a program written in MC-Basic or directly from the command line.

The parameters can be passed by value as well as by reference. This function returns a value that can be processed by an MC-Basic application.

MC-BASIC provides a facility to load a user object-file into the RAM of the MC. This file is relocatable and must include information about global symbols for the MC to find the C-function. The object module may consist of any number of functions. The only restriction is RAM limit.

### 2.5.1. Object Files

Object files (extension ".o"), contain compiled C-programs and are stored in the Flash disk. You can **SEND** object files to the controller, load files into RAM with **OLOAD** and unload them with **OUNLOAD**. If **OLOAD/OUNLOAD** fails for any reason, details are found in the OLOAD.ERR file.

## 2.5.2. Prototype File

For the MC-Basic language translator to match function parameters, provide a C-function prototype file (PROTO.PRO). It is important to understand that matching between the provided MC prototype and actual C implementation cannot be tested by the translator. ***It is your responsibility to keep consistency between the function prototype and C implementation.***

**NOTE**

***To speedup translation of the prototype file, PROTO.PRO is copied into RAM at startup and at every OLOAD. When PROTO.PRO is modified and sent to the controller, issue an OLOAD or reboot the MC to refresh the prototypes in the RAM.***

The translator is case-insensitive, while object module loader is case-sensitive, so ***write the name of a C function in capital letters within the C code***. MC prototypes of C functions and C function calls through the MC are not case sensitive.

Parameters can be passed "by value" and "by reference"
Few object files may be incrementally linked together.

The general syntax for the MC prototype of a C-function is:

IMPORT_C <*Function_Name*> ({AS <*type*>}, {BYVAL AS {<*type*>}) {AS <*type*>}

A parameter passed by value has the BYVAL prefix.
A parameter passed by reference has no prefix.
Prototypes do not include parameter names.
A C function prototype with no parameters is written with empty parentheses:

IMPORT_C <*Function_Name*> ( ) {AS <*type*>}

A C function prototype with no returned value (a void function) is written as:

IMPORT_C <*Function_Name*> ({AS <*type*>}, {BYVAL AS {<*type*>})

A C-function accepts any combination of double-, long- and string-type parameters. The parameters may be passed "by value" or "by reference".

The returned value may be long, double, string or none (for C functions with void returned value).

Examples of prototypes and implementation:

| MC-Basic Prototype | C Implementation |
|---|---|
| import_c cFunc_LV As Long | int CFUNC_LV(void); |
| import_c cFunc_DV As double | double CFUNC_LV(void); |
| import_c cFunc_SV As string | char* CFUNC_LV(void); |
| import_c cFunc_VV() | void CFUNC_VV(void); |
| import_c cFunc_LL(ByVal As Long) As Long | int CFUNC_LL(int L); |
| import_c cFunc_DRD(As Double) As Double | double CFUNC_DRD(double* D); |
| import_c cFunc_SRS(As String) As String | char* CFUNC_SRS(char** S); |
| Import_C cFunc_LARD**(**[*] as Long) as Double | double CFUNC_LARD(long *l); |
| import_c cFunc_SS(ByVal As String) As String | char* CFUNC_SS(char* S); |

Parameters can be of type long, double or string; in any order, up to 32 parameters.

Only one-dimensional arrays are allowed as C-functions arguments.

Example of calling a C-function from MC-Basic:

```
Dim shared Str2global as string
Dim shared Str1global as string
Dim shared LongArray[100] as long
Dim shared d1 as double
program
'   Use OLOAD to load the object file before loading the program
'   that uses the C-Functions

'   /* No Parameters */
'   int CFUNC_LV(void)
   ?CFUNC_LV()

'   double CFUNC_DV(double)
   ?CFUNC_DRD(1.2345)

'   /* returns string */
   char* CFUNC_SV(void)
   ?CFUNC_SV()

'   /* No Parameters, not return value */
'   void CFUNC_VV(void)
   CFUNC_VV()
'   /* Strings */
'   char* CFUNC_SS(char* S)
        Str2global=CFUNC_SS(Str1global)

'   /* Arrays */
'   double CFUNC_LARD(long *l);
        d1= cFunc_LARD(LongArray)
end program
```

Example of PROTO.PRO:

```
'No Parametrs
import_c CFUNC_LV() As Long
import_c CFUNC_DV() As Double
import_c CFUNC_SV() As String
import_c CFUNC_VV()
'A Single "By Value" Parameter
import_c CFUNC_LL(ByVal As Long) As Long
import_c CFUNC_DD(ByVal As Double) As Double
import_c CFUNC_SS(ByVal As String) As String
'A Single "By Reference" Parameter
import_c CFUNC_LRL(As Long) As Long
import_c CFUNC_DRD(As Double) As Double
import_c CFUNC_SRS(As String) As String
'Multiple "By Value" Parametrs
import_c CFUNC_LVALP(ByVal As double, ByVal As Long, ByVal As
String) As Long
import_c CFUNC_VVALP(ByVal As String, ByVal As double, ByVal AS
Long)
'Multiple "By Reference" Parameters
import_c CFUNC_DREFP(As String, As Long, As Double) As Double
import_c CFUNC_VREFP(As Long, As Double, As String)
' Mixed Parameters
import_c CFUNC_VMIXP(ByVal As Long,As Double,ByVal As String,As
String,As Long,ByVal As double) AS Long
```

Example of test.c:

```
/* No Parameters */
int CFUNC_LV(void)
{
   return 1;
}
double CFUNC_DV(void)
{
   return 2.2;
}
char* CFUNC_SV(void)
{
   return "SV";
}
void CFUNC_VV(void)
{
   int fd = open("/tyCo/1",2,0);
   fdprintf(fd,"VV\r\n");
   close(fd);
}
/* A Single "By Value" Parameter */
int CFUNC_LL(int L)
{
   L = L + 1;
   return L;
}
double CFUNC_DD(double D)
{
   D = D + 2.2;
   return D;
}
char* CFUNC_SS(char* S)
{
   strcpy(S,"SS");
   return S;
}
/* A Single "By Reference" Parameter */
int CFUNC_LRL(int* L)
{
   *L = *L + 3;
   return *L;
}
double CFUNC_DRD(double* D)
{
   *D = *D + 4.4;
   return *D;
}
char* CFUNC_SRS(char** S)
{
   strcpy(*S,"SRS");
   return *S;
}
/* Multiple Parameters */
/* By Value Parameters */
int CFUNC_LVALP(double D, int L, char* S)
```

***Continued on next page…….***

```
{
   return (D + L + atoi(S));
}
void CFUNC_VVALP(char* S, double D, int L)
{
   int fd = open("/tyCo/1",2,0);

   fdprintf(fd,"Original Values:%d, %f, %s\r\n", L, D, S);

   L = 10;
   D = 22.2;
   strcpy(S,"VValP");

   fdprintf(fd,"New Value:%d, %f, %s\r\n", L, D, S);

   close(fd);
}

/* By Reference Parameters */
double CFUNC_DREFP(char** S, int* L, double* D)
{
   return (*D + *L + atof(*S));
}
```

## 2.5.3. Special Considerations

Motion and System properties can only be passed by value.

**KILLTASK** will not unblock a task locked inside the C-function. An endless block on a semaphore, message queue, etc. inside a C-function prevent sa task from correct being killed. If C-function has some blocking operation system call (**taskDelay**(), **semTake**(), **msgQGet**(), **msqQSend**(), IO operation, etc.), this may interfere with killing the user task.

When an object module is loaded several times without unloading, all its instances are kept in the RAM, while the system symbol table has references only to the resent instance. MC-Basic applications may refer to the obsolete version of some modified object file.

**OUNLOAD** does not check to see if the object file is in use. It is your responsibility to ensure that the object is not unloaded as long as there tasks and libraries that refer to the object.

Consider following example, both the program TASK1.PRG and TASK2.PRG use the same function from object module.

```
→oload my_obj.o
→load TASK1.PRG
→send my_obj.o        ‘ send updated version of object file
→oload my_obj.o
→load TASK2.PRG
```

In this example, TASK1.PRG references the oldest instance of my_obj.o while TASK2.PRG references the recent version.

# 2. 6 SEMAPHORES

Semaphores are the basis for synchronization and mutual exclusion. The difference is that the mutual exclusion semaphore is created as “full” or “1”, while the synchronization semaphore is empty “0”. If the semaphore is used for protecting mutual resources, it is taken before accessing the resource and releases at the end. A synchronization semaphore is given by the producer and taken (consumed) by the consumer.

***A semaphore is created as “full.”***

**NOTE**

Global semaphore are defined with **COMMON SHARED** in CONFIG.PRG or from the command line. Since a semaphore's purpose is to protect data among tasks, there is no meaning to local semaphores.

A semaphore is *given/released* by **SEMAPHOREGIVE** and *taken/consumed* by **SEMAPHORETAKE**. It is possible to specify a time out of up to 5000 ms. **SEMAPHORETAKE** acquires a semaphore and returns before timeout or does not acquire a semaphore and returns after timeout.

***Mutual exclusion semaphores are taken and given by the same task. Synchronization semaphores are given by one task and taken by another task.***

**NOTE**

## 2.6.1. Mutual Exclusion Semaphores

Mutual exclusion semaphores lock resources by taking a semaphore. Another task(s) competing for the same resource is blocked until the semaphore is released.

Example of a mutually exclusive semaphore:

```
‘ common shared comMutex as semaphore
‘ defined in config.prg
Program
Open COM2 BaudRate=9600 Parity=0 DataBits=8 StopBit=1 As #1
While 1
‘ take semaphore lock serial port
   If SemTake(comMutex,5000) = 1 Then
      Print #1,”Hello ”;
      Print #1,”world”
      SemGive(comMutex)   ‘unlock serial port
   End if
   End While
End program
```

## 2.6.2. Synchronization Semaphores

Synchronization semaphores are essential in producer-consumer applications where task A prepares some data, while task B consumes it. In this case, the semaphore may eliminate constant polling for ready data and save considerable CPU resources.

Example of Producer:

```
‘ common shared syncSemaphore as semaphore
‘ defined in config.prg
‘ common shared globalA as long       ‘ defined in
config.prg
Program
Dim dummy as long
‘ Semaphore is created as “full” - flush it before first use
dummy=semTake(syncSemaphore)           ‘ no waiting
While 1
   globalA=globalA+1                   ‘ produce some data
   semGive(syncSemaphore)
   sleep (100)
   End While
End program
```

Example of Consumer:

```
' common shared syncSemaphore as semaphore
' defined in config.prg
' common shared globalA as long     ' defined in config.prg
Program
While 1
   If SemTake(syncSemaphore,5000) = 1 Then
      Print "A is "; globalA
   End if
   End While
End program
```

## *2. 7 VIRTUAL ENTRY STATION*

**V**irtual **E**ntry **S**tation (VES) allows a CLI-like approach to the system from any task or library. VES permits access to virtually any system property. The application developer must take into account that, in the contrast to regular programs and libraries, the string is passed to VES and are interpreted and translated at run-time. Execution takes much more time than the equivalent line in the program.

The response of VES is ASCII an string, which could be immediately printed or assigned to any string variable. Since VES involves translation and then interpretation, the result could also be an error message. To distinguish between a normal response and an error, VES gives either a “D:” or “E:” prefix to the output.

VES Example:

```
Program
Dim s2 as string
Dim s1 as string
S2="?sys.time"
S1=VesExecute(s2)
Print s1
S1=VesExecute("sys.time") ' incorrect, returns syntax error
Print s1
End program

Output:
D:18:27:19
E:Error:  7039, "Syntax Error", Module: Translator
```

## 2. 8 OUTPUT REDIRECTION

Typically, tasks inherit their standard output from the loading context, such as the entry station or standard output of a loader task. Sometimes, you must redirect task output (print) to a different place, such as another entry station or even a virtual entry station. **RedirectStdOut/RedirectStdOut$** provides this capability.

```
-->RedirectStdOut task32.prg NewStdOut=1
-->
```

NewStdOut may be:

1 – Ethernet
2 – DPRAM
3 – Virtual Entry Station
4 – Etherent2

***Systems with FPGA version 8.51, firmware 4.0.0 and higher support IP over DPRAM.*** NewStdOut ***type is 1,3 or 4.***

**CAUTION**

## 2. 9 FILE OPERATIONS

Input and output, whether to or from files, are supported on Flash and RAM devices. The files are accessed either in serial or random order. Currently, only ASCII text files are supported – both printable and non-printable characters.

### 2.9.1. OPEN

Opens an existing file or creates a new one with the name specified in the string expression. The file name should not exceed 8 characters plus extensions. Acceptable file extensions are:

PRG , DAT , TSK , CMP for permanent files on the Flash disk
REC for temporary files on the RAM disk

You can open a text file to read, write or append to the existing file according to mode flag.

"r" - open text file for reading
"w" - truncate to zero length or create text file for writing
"a" - append; open or create text file for writing at end-of-file

Use **APPEND** to add new lines at the end of the original contents of the file.Use **WRITE** to overwrite the previous file or to create a new file.

## 2.9.2. OPEN #, INPUT #, CLOSE, LOC

See Serial port.

## 2.9.3. TELL

Returns current file pointer, offset from the beginning of the file. Value has meaning only for **SEEK** to move the file pointer within a file.

## 2.9.4. SEEK

Moves file pointer to specified location. Use with **TELL**.

File operations example:

```
program
    dim pos as long
    dim d1 as double
    dim s1 as string
'create file for writing
    Open "test.prg" mode ="w" as #1
    print #1, "1.2345"
    close #1
'append to existing file
    Open "test.prg" mode ="a" as #1
    print #1, PI
    close #1
'read from file
    Open "test.prg" mode ="r" as #1
'how many characters in the file?
    print "File contains ";loc(1);" characters"
    s1 = input$(#1)
'convert from string to double
    d1=val(s1)
    ?d1
'save current file pointer
    pos = tell(#1)
    ?input$(#1)
'rewind file
    seek (#1,pos)
    ?input$(#1)
    close #1
end program

Output:
File contains 31 characters
1.234500000000000e+00
3.141592653590000e+00
3.141592653590000e+0
```

# 3. PROJECT

A project is all the software written for an application. The projects in MC-BASIC are multi-tasking. They 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 that run simultaneously with many other tasks. Projects also include other files such as cam tables and record files. Projects are controlled from the BASIC Moves (BMDS) software. BMDS automatically sets up a separate directory for each project. For detailed information on any commands listed within this section, refer to the *SERVO**STAR**® MC Reference Manual.*

For clarity, a project is managed by BMDS and conveniently handles a group of logically coupled files (programs, CAM tables, etc.), while the MC deals with separate files (PRG, CAM, etc.). The Motion controller does not keep project files in the Flash and it is not aware of logical coupling of files and programs.

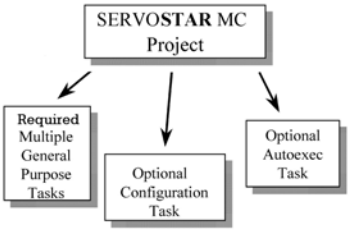
## *3. 1 PROJECT STRUCTURE*

Project files include tasks and cam tables. Each project can contain three types of tasks:

**General-purpose task**

An **optional configuration task** (Config.Prg) to declare groups, programmable limit switches (PLS), cams, global variables and load of global libraries.

An **optional autoexec task** (Autoexec.Prg) to automatically start the application on power-up.

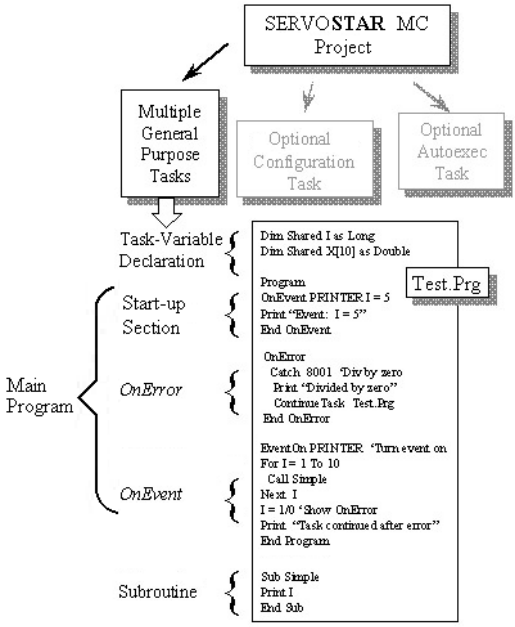



## *3. 2 TASKS*

The three types of tasks are general-purpose tasks, configuration tasks, and autoexec tasks. Each type of task is outlined below.

### 3.2.1. General Purpose Tasks

General-purpose tasks are the work horse of the MC language. They implement the basic logic of your application. The great majority of your programming is implemented with general-purpose tasks. Each general-purpose task is divided into three sections: a task-variable section, a main program, and subroutines. The main program is further divided into three sections: a Start-up section, an OnError section, and an OnEvent section.




**A task-variable definition section**
The task-variable definition 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…End Program keywords. The main program itself has three sub-sections:

**The Start-up section**
The start-up section immediately follows the Program keyword. This is where task execution begins when you start a task.

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

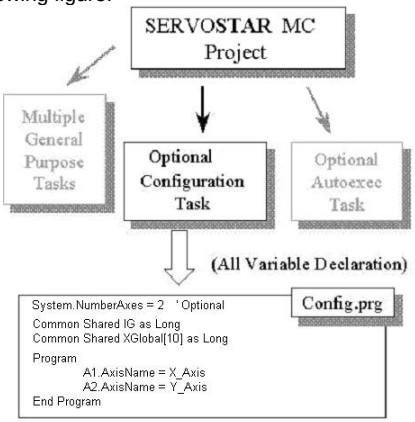
**OnEvent section**
This section contains optional blocks of code that respond to realtime changes, such as a motor position changing or an input switch turning on. The main program can contain code to automatically respond to events. This reduces the effort required to make tasks respond quickly and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

Event handlers begin with the OnEvent and end with End OnEvent keywords. One OnEvent…End OnEven keyword combination is required for each realtime event. Event handlers must be contained wholly within the main program.

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

## 3.2.2. Configuration Task

The name of the configuration task is Config.Prg. The configuration task is used for declaration of number of axes, 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. Refer to the following figure.



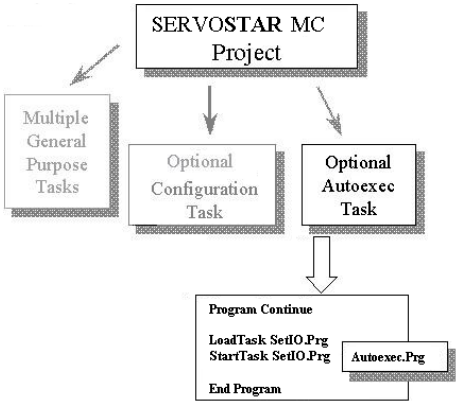

The configuration task can contain a program. Axes can be renamed here.

## 3.2.3. AutoExec Task

The AutoExec task (AutoExec.Prg) 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 the outputs to the desired starting values. That way, the outputs are set immediately after the MC boots, usually sooner than the PC.

For safety considerations we do not recommend starting of motion from the AutoExec task. Motion should be started by explicit operator's request either by I/O or communication link from host PC.

Set the AutoExec task to run on power up. To do this, add the keyword Continue to the Program command. Do not include OnError or OnEvent sections in the AutoExec. Limit the AutoExec task to starting other tasks in the system. Refer to the next figure.




## 3. 3 PROGRAM DECLARATIONS

You must declare the start of programs and subroutines. For programs, use the Program…End Program keywords. Use Sub…End Sub keywords for subroutines.

The Program…End Program keywords mark the boundary between the variable declaration section and the main program. Each task must have only one Program keyword and end with the End Program keyword.

```
Program            'Standard form of program command
     <code for program>
End Program
```

The AutoExec task, which must be loaded and run automatically at power-up must have **CONTINUE** following Program.

You pass parameters (either scalar or array) to the subroutine, which can then be used in the code of the subroutine. In the declaration line for the subroutine (SUB<*name*>), you declare the variable names and types of the parameters to pass. Parameters are passed either by reference or by value (ByVal). The default method is to pass parameters by reference. Arrays are passed only by reference. When you pass a variable (whether the variable is local to the task or is global) by reference, you pass the address of the variable to the subroutine, which changes the value of the variable (if the code of the subroutine 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 subroutine. The subroutine cannot change the value of the local variable. The syntax for defining a subroutine is:

SUB <*name*> ({{ByVal} <*p_1*> as <*type_1*> }…{, {ByVal} <*p_n*> as *type_n*>})
  { *local variable declaration* }
  { *subroutine code* }
END SUB

There is no explicit limit on the number of subroutines allowed in a task. All subroutines must be located following the main program, and must be contained wholly outside the main program. Subroutines can only be called from within the task where they reside. Subroutines may be recursive (call itself).

Use **CALL** to execute a subroutine:

```
CALL <subroutine>({<p_1>...<p_n>})
```

where:

<*subroutine*> is the name of the subroutine
<*p_1*>...<*p_n*> are the subroutine parameters

**Parentheses are not used in a subroutine if no parameters are passed.**

MC-BASIC automatically checks the type of compliance between the subroutine declaration and the subroutine call. Any type mismatch causes an error during program loading. Automatic casting applies to numericc variables types. For example, suppose MySub is a subroutine that takes a Long parameter. In this case, the following scenario applies:

```
CALL MySub(3.3432)
'OK: The double value 3.3432 is demoted to the Long value, 3
CALL MySub(a)
'Error: a is a string, there is a type mismatch
Call MySub(3,4)
'Error: The number of parameters is not correct
```

See the Subroutine Example in Appendix A for further details.

## 3.3.1. Arrays

Arrays can only be passed by reference. If a user tries to pass a whole array by value, the translator gives an error. Array syntax is:

SUB <*name*> ({<*p_1*>([*])+ as <*type_1*>}…{, <*p_n*>([*])+ as <*type_n*>})
{ *local variable declaration* }
{ *subroutine code* }
END SUB

where

<*p_l*> : name of array variable
<*p_n*> : name of array variable
[*] : dimension of array without specifying the bounds
+ : means one or more

Syntax example:

```
SUB mean(x[*][*] as DOUBLE, TheMean[*] as LONG)
 DIM sum as DOUBLE
 DIM I as LONG
  FOR i = 1 to 100
     sum = sum + x[i][1]
  NEXT i
  TheMean[1] = sum/100
END SUB
```

**Subroutine Libraries**. As you develop programs for a variety of applications, you will find that there are some subroutines that you use repeatedly. 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 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 MC 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 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. 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 is 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.

```
LOAD MyLibrary.lib
```

To unload the library, enter:

```
UNLOAD MyLibrary.lib
```

If a program or library task references a subroutine in a library, the program or 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. **IMPORT** must be placed at the beginning of the program code before any other command or declaration.

```
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 (because you can not import two library files that contain the same library function name).

**User-defined Functions**. MC BASIC allows you to define your own functions to be used in programs in the same manner as using BASIC's pre-defined functions. User-defined functions are composed with multiple lines and are recursive (can call 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 funtions return a value, you can use functions expressions (i.e., 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. These parameters are then used in the code of the function. Declare the variable names and types of parameters you want to pass in the declaration line for the function. Parameters can be passed by reference or by value (ByVal). The default is by reference. When you pass a variable by reference, you actually pass the address of the variable to the function, which changes the value of the variable (if the function code is written to do this). When you pass a variable by value, a copy of the value of the variable is passed to the function. The function cannot change the value of the variable. Arrays are only passed by reference.

To set up the return value, assign the value you want to return to a virtual variable with the same name as the function somewhere in the code of the function. You do not declare the variable, but the function uses it 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 of the main program.

A function can be recursive (can call itself). The following example defines a recursive function to calculate the value of *N*:

```
FUNCTION Factorial (ByVal N As Long) As Double
'Declaring N as long 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 a recursive function, you must have an IF statement to force the function to return without the recursive call being executed. Otherwise, the function never returns once it is called.

## 3. 4 MULTI-TASKING

The MC supports multi-tasking. You can have multiple tasks running independently, sharing a single computer. A task is a section of code that runs in its own context. Microsoft Windows® is a multi-tasking system. If you open Explorer and Word at the same time, they run nearly independently of each another.

In this case, both Explorer and Word have their own contexts. They share one computer, but run as if the other were not present. 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.

With MC-BASIC, 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, and you can prescribe interactions between tasks.

Multi-tasking is used when you want multiple processes to run largely independent of each other. For example, if you are using the MC to interface to the operator, you will usually use a separate task to execute the interface code. Another example is when two parts of a machine are largely independent of each other. There is usually some control required between tasks as one task may start or stop another.

If a machine is simple to control, you should try to keep the entire program in one task (in addition to Config.Prg). If you do need to use multi-tasking, you should keep a highly structured architecture. Kollmorgen recommends that you limit use of the main task for axis and group set up, machine initialization, and controlling the other tasks. Normal machine operation should be programmed in other tasks. For example, Main.Prg might be limited to setting up axes, and then starting Pump.Prg, Conveyor.Prg, and Operator.Prg.

Do not split control of an axis or group across tasks. You can put control for multiple axes in one task. Ideally, you should use multiple tasks for machines where different sections operate more or less independently. You can also use tasks to implement different operational modes.

Multi-tasking 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 is difficult to understand and recreate. Limit the use of tasks to situations where they are needed.

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. Add the event handler to an existing task to respond to an event.

Do 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 multi-tasking and when to avoid it requires some experience. If you are new to multi-tasking, you may want to start slow until you are familiar with how it affects program structure. When you start a new project, BASIC Moves 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 BASIC Moves tool bar.

## 3.4.1. Loading the Program

BASIC Moves 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 “Load Task”and “Run Task” buttons on the tool bar. By default, tasks are loaded from the host PC to the MC 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 Manager->Options and then changing the priority in the bottom of the window. If you structure your software so that the main program loads all other tasks, the Run Project button starts your machine.

## 3.4.2. Preemptive Multi-tasking & 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.

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 that runs at a medium priority and perhaps a background task that runs at a low priority, with a few high priority tasks. At every time slice, the MC re-evaluates which task has the highest priority and assigns resources to it.

The BASIC Moves terminal window relies on the MC command line task, which runs at priority 2. If you start a task with priority 1, the terminal will not be available until the task is complete or idle. Consequently, you will not be able to communicate with the MC and you may have to power-down the system to recover the terminal window. You can optionally 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** is a method by which the operating system divides up resources when multiple tasks share the same priority level. The MC 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 duration. This method is sometimes called round robin scheduling.

## 3.4.3. Inter-Task Communications and Control

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. For detailed information on these commands, refer to the *SERVO**STAR**® MC Reference Manual*.

**StartTask** starts tasks from the main task. For testing, you can use **STARTTASK** from the terminal window. DanaherMotion recommends you do not use **STARTTASK** in AutoExec.Prg. The syntax of **STARTTASK** is:

StartTask <*TaskName*> {Priority = <*Level*>}{NumberOfLoops = <*Loop Count*>}

**NOTE**

***NOL is a short form for NumberOfLoops.***

where:

<*Level*> is a long with value between 1 and 16. If <*Level*> is not entered, it defaults to 16, the lowest priority. Priority = 1 is the highest priority.

<*Loop Count*> is either -1 (indicating unlimited number of loops) or between 1 and 32768 indicating the number of times the task is executed. If <*Loop Count*> is not entered, it defaults to 1.

For example:

```
StartTask Task1.Prg Priority=8 NumberOfLoops = -1     'Run Task1 forever
StartTask Main.Prg NOL=1                              'Run Main once
```

**IdleTask** stops the task at the end of the line currently being executed and idles all its events. An idled task can be continued (using **CONTINUETASK**) or terminated (using **KILLTASK**). **IDLETASK** does not stop motion currently being executed. This is significant because all the events are idled and cannot respond to an axis' motion. Tasks can be idled explicitly by other tasks, but cannot idel itself. This command is issued from a task or the terminal window. The syntax of **IdleTask** is:

IdleTask <*TaskName*>

For example:

```
IdleTask TASK1.PRG
```

Tasks that have been idled with **IDLETASK** are restarted only with **CONTINUETASK**. The command continues an idled task from the point at which it was stopped, or continues task execution after a break point has been reached. It is frequently used to restart tasks from the ONERROR error handler. If a run time error has occured, CONTINUETASK retries the line which caused the error. The error must be corrected before the task continues. This command is issued from any task or the terminal window. The syntax of **ContinueTask** is:

ContinueTask <*TaskName*>

For example:

```
ContinueTask TASK1.PRG
```

**KillTask** aborts the execution of a task. The program pointer is left on the line at which the task was stopped. The first action of **KILLTASK** is to kill and delete all events associated with the task. This is done to ensure that no event initiates an action after **KILLTASK** was executed. **KILLTASK** is issued from the terminal window. The syntax of the **KillTask** is:

KillTask <*TaskName*>

For example:

```
KillTask TASK1.PRG
```

## 3.4.4. Monitoring Tasks From the Terminal

For detailed information on these commands, refer to the *SERVO**STAR**® MC Reference Manual*.

**TASK.STATUS** provides the current state of any task. You can query **TASK.STATE** from the terminal window. You cannot use **TASK.STATUS** from within a program. The syntax for **TASK.STATUS** is:

? <*TaskName*>.Status

For example:

```
? TASK1.PRG.Status
```

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

For example, if you type:

```
? TaskList
```

A typical result is:

```
TaskName = TASK1.PRG, Status = sstep, Priority=16
TaskName = DO.PRG, Status = suspend, Priority=4
```

## 3.4.5. Relinquishing Resources

When tasks of different priorities compete for processor time, the highest priority task always 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 under which a task relinquishes resources: when the task is terminated, when the task is suspended, or when the task is idled. For detailed information on these commands, refer to the *SERVO**STAR**® MC Reference Manual*.

A **task terminates** when it is finished executing. If a task starts with **NUMBEROFLOOPS** greater than zero, the task executes the specified number of times and terminates. The task relinquishes all resources. Terminated tasks remain loaded in the system and can be restarted.

One task can terminate another task by issuing **KILLTASK**. A task relinquishes all resources after the kill command. Killed tasks remain in the system and can be restarted.

Tasks relinquish processing resources temporarily when they are **suspended**. A task is suspended when it is waiting for a resource or is delayed. Suspended tasks still monitor events as long as the event priority is higher than the task priority. Never run a task at a higher priority level than any of its events.

Use **SLEEP** to delay a task for a specific period of time. This command can only be issued from within the task. One task cannot issue a **SLEEP** for another task. **SLEEP** causes the task to relinquish resources until the sleep time has expired.

**Idled tasks** relinquish resources. In this case, resources are relinquished until another task revokes the idle by issuing a **CONTINUETASK**.

**Delete Task/Library** deletes a file from the Flash Disk. Only filesnot loaded into RAM can be deleted.Files that are protected by a password may not be deleted. For example:

```
Delete FILE1.PRG
```

# 3. 5 EVENT HANDLER

The main program can contain sections which 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** and occur just after the **Program** keyword.

After **OnEvent** is loaded, turn the event On with **EventOn** just after the **End OnEvent** keyword (enable immediately). However, you can enable and disable **OnEvent** at any time using **EventOn** and **EventOff**. Multiple **OnEvents** can be written sequentially. The MC system can support 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).

It is important to understand that **OnEvent**s are different from ordinary tasks. **OnEvent**s are preemptive within the task. That is, an **OnEvent** runs until complete and the 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, **OnEvents** are 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. The rules are valid for a single process, (that is, the parent task and its events), while events of the respective tasks in the system share the CPU among themselves.

## 3.5.1. OnEvent

The syntax of **OnEvent** is:

OnEvent [*EventName*] [*Condition*] {Priority=*EventPriority*}{ScanTime=*time*}

where

*EventName* is any legal name that is otherwise not used in the task.

*Condition* is any logical expression such as System.Dout.1 = 1. The event fires on transitions of the condition from false to true.

*Priority* is an integer from 1 (highest) to 16 (lowest). If not entered, *priority* defaults to 1. The priority should always be higher than the priority of the task or the event never runs.

*Time* is an integer indicating the number of cycles between each scan. *Time* defaults to 1.

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

```
OnEvent MOVE_ON_TRIGGER System.Din.1=ON
    Move X-axis 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 resources until it is complete.

*ScanTime* is in cycles of the SERCOS update rate. This is normally 2 or 4 milliseconds. Setting *ScanTime* to 5 configures the system to check the event condition every 10 or 20 milliseconds, depending on your update rate. For example:

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

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

## 3.5.2. EventOn

**EventOn** enables **OnEvent**. The syntax of **EventOn** is:

EventOn [*Event Name*]

**EventOn** must come after the definition of the **OnEvent**.

## 3.5.3. EventOff

**EventOff** disables **OnEvent.** The syntax of **EventOff** is:

EventOff [Event Name]

Refer to the *SERVO**STAR**® MC Reference Manual* for information additional information about **OnEvent**, **EventOn**, and **EventOff**.

## 3.5.4. EventList

**EventList** provides a complete list of all events in all tasks with their name, the task name, the priority, whether the event is armed, and current status. **EventList** is valid only from the terminal window. For example:

```
? EventList
```

the result is something like the following line for each task:

Name = IOEvent Owner=Task1 Edge=1 Status=1 Scan=1 Priority=5
Action = Stop

where:

*edge=1* indicates the event is armed (that is, the condition is false so that the condition becoming true will fire the OnEvent)

*status=1* means the event is enabled

## 3.5.5. EventDelete

Deletes the specified event. The event does not respond to the specified condition until the task is executed again and the event is enabled.

```
EventDelete EVENT1
```

## 3.5.6. Events at Start-up

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

## 3.5.7. Program Flow and **OnEvent**

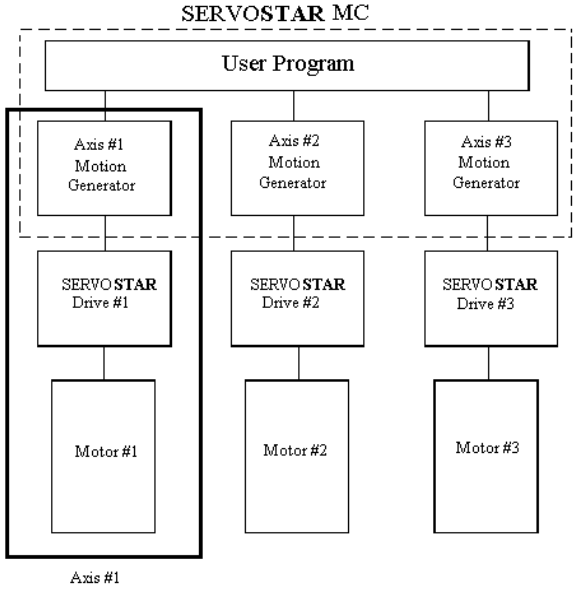
You can use **GoTo** commands within an **OnEvent** block of code. However, because **OnEvent** interrupts the main program, you cannot use **GoTo** to branch out of the event handler or into the event handler. You cannot place **OnEvent…End OnEvent** in the middle of program flow commands (e.g., **For…Next**,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

# 3. 6 SETTING UP AXES

In this section, we set up the axes in the system. We will discuss units for position, velocity, and acceleration. We discuss how the SERVO**STAR** MC's acceleration profile works. We talk about how to set up limits for axes. We conclude with a discussion of a few advanced topics: an overview of SERCOS, simulated axes, and dual-loop position control.

## 3.6.1. Axis Definition

The MC is oriented around axes. An axis is a combination of a drive, a motor, and some portions of the MC. The diagram below shows an MC axis:




Each axis has a set of properties that are defined by the MC. These properties are used in the calculation of position and velocity commands, monitoring limits, and stores configuration data for the axis. The MC sends commands to the drive and the drive controls the motor. All these components together form an axis of motion.

## 3.6.2. Axis Name

The MC automatically sets up all your axes according to their addresses: A1, A2, A3, and so on until A32 ( the maximal number of axes is limited by User Autorization Code by manufacturer ). You access axis properties by preceding the property name with the axis name and a period. For example, each axis has a property, **VELOCITYFEEDBACK**, which provides the current velocity of that axis. You can query the **VELOCITYFEEDBACK** value with the **?** command in the BASIC Moves terminal:

```
? A1.VelocityFeedback
```

You can rename the axis. It is usually worthwhile to name the axes according to their function. It makes your program easier to understand and to debug. For example, you could enter:

```
A1.AxisName = ConveyorAxis
```

Later, you can query the velocity with:

```
? ConveyorAxis.VelocityFeedback
```

The axis name may only be changed into the configuration program Config.Prg. You cannot print the axis name at any time.

## 3.6.3. Drive Address

You must specify the drive address property of the axis. Simulated axes do not have drive addresses and do not require this property to be assigned. This assigns the physical address (dip switch setting on the SERVO**STAR** drive) to the axis defined in the SERVO**STAR** MC. If we wanted to assign the drive with address 5 to axis 1, we type the following command into the terminal window of the BASIC Moves Developer Studio or in our program:

```
A1.Dadd = 5          ' Assign drive address 5 to axis 1
```

The short form, **DADD**, of axis property **DRIVEADDRESS** is used in the example. **Axis.DriveAddress** property should meet to the drive hardware address configuration . Most axis properties and constants have short forms that can speed typing and ease the programming effort. Refer to the *SERVO**STAR**® MC Reference Manual* for more information.

## 3.6.4. Starting Position

Before we get started here, you must:

**Install and tune drives**

Follow the process for installing and tuning all drives according to the *SERVO**STAR**® MC Installation Manual*. Run **MOTIONLINK**® to select your motor type and tune the axis to your satisfaction.

**Install and check out your MC**

Install and check out your SERVO**STAR** MC according to the *SERVO**STAR**® MC Installation Manual*. Wire all I/O, connect the fiber-optic ring, and follow the procedures in the *SERVO**STAR**® MC Installation Manual* to check out the wiring.

**Install BASIC Moves Development Studio**

Install BASIC Moves Development Studio after reviewing the *BASIC Moves User's Manual.* We use the terminal screen here extensively. Remember, any commands typed in the terminal window are lost when you power down. Enter these commands in your MC program to be recalled at power up.

## 3.6.5. BASIC Moves Auto Setup Program

BMDS automatically generates a setup program each time you open a new project. BMDS first requests information regarding units and preferences for each axis in the system. Then, it generates a program with most of the setup for your application.

The auto setup program has four sections: a short task-variable declaration, an example program which generates simple motion, a SERCOS setup subroutine, and an axis setup subroutine. The variable declaration section provides one or two examples of variable declarations for a task.

The motion program cycles axis A1 10 times. Most of this main program is commented out because this code enables the drive and generates motion commands. Remove the single-quote comment markers for the program to run. These lines are commented out because to ensure that it is safe to operate your machine before executing this program. This includes tuning your axes properly to assure stable motion control. Refer to the *SERVO**STAR**® MC Installation Manual* for more information.

The third section of the auto setup program is a SERCOS setup subroutine. This subroutine brings up the SERCOS ring in preparation for the drives being enabled. The final section of the auto setup program is the axis setup subroutine. This subroutine loads axis properties for units and limits. Refer to Appendix A for a Sample Nesting Program.

## 3.6.6. User Units

The MC supports user units with one floating-point conversion factor for each of six types of properties: position, velocity, acceleration, jerk, external position, and external velocity.

<***axis***>**.DIRECTION** is applied as a multiplier to the units conversion factors. The movement factors (*i.e.*, **POSITIONFACTOR**, **VELOCITYFACTOR**, **ACCELERATIONFACTOR**) are only assigned positive values and **DIRECTION** determines their sign. **DIRECTION** can take one of two values: -1 for forward direction or +1 for reverse direction.

MC units allow you to work in units that are convenient. All unit types are independent, so position units are inches, velocity units are feet/min and acceleration is rpm/second. Units are changed only when the axis is disabled.

## 3.6.7. Position Units

Position units are controlled by **POSITIONFACTOR** or **PFAC**. The MC internal units are encoder counts for encoder-based systems or 65536 counts per revolution for resolver-based systems. Encoder counts are four times encoder lines. To set up position units, determine how many counts are in one of your selected units. For example, suppose you had this system:

Desired units: inches
Rotary motor (as opposed to linear motor)
2000 line Encoder
3:1 gearbox
5 turn per inch ball screw

Determine **POSITIONFACTOR** (number of counts per inch):

1 inch = 5 turns on the screw
= 15 turns into the gearbox
= 15 * 2000 lines
= 15 * 2000 * 4 counts
= 120,000 counts

**POSITIONFACTOR** is set to the number of counts per inch: 120000.

```
A1.PositionFactor = 120000
```

For a second example, let's change the first example from encoder to resolver, change the ball screw and use meters for the units:

Desired units: meters
Rotary motor
Resolver (65536 counts per revolution)
3:1 gearbox
2 turn/cm ball screw

Determine **POSITIONFACTOR** (number of counts per meter):

1 meter = 200 turns on the screw
= 600 turns into the gearbox
= 600 * 65536 counts
= 39,321,600 counts

**POSITIONFACTOR** is set to the number of counts per meter: 39321600. If you use the BASIC Moves auto setup program, provide the motor resolution and number of motor rotations per unit movement and BASIC Moves calculates the math.

## 3.6.8. Velocity Units

Velocity units convert MC internal velocity units (counts/mSec) to your units. They are controlled using **VELOCITYFACTOR** or **VFAC**. Normally, velocity units are either the position units per second or per minute, or they are rpm. If you want velocity units to be your position units per second, set **VELOCITYFACTOR** to **POSITIONFACTOR** divided by 1000 (to convert milliseconds to seconds):

```
A1.VelocityFactor = A1.PositionFactor/1000  ' for position units/sec
```

If you want position units per minute, divide by 60000:

```
A1.VelocityFactor = A1.PositionFactor/60000  ' for position units/min
```

If you want rpm, you must calculate the number of counts per revolution and then divide that by 60000 (to convert milliseconds to minutes):

Desired units: rpm
2000 line Encoder

You first need to determine the number of counts per inch for each revolution:

1 rev = 2000 lines
= 2000 * 4 counts
= 8000 counts

**VelocityFactor** is set to the number of counts per revolution divided by 60000:

```
A1.VelocityFactor = 8000/60000
```

## 3.6.9. Acceleration Units

Acceleration units convert MC internal acceleration units (counts/msec$^2$) to your units. They are controlled using <***axis***>**.ACCELERATIONFACTOR** or <***axis***>**.AFAC**. Normally, acceleration units are either the velocity units per second or per minute. If you want acceleration units to be your velocity units per second, set **ACCELERATIONFACTOR** to **VELOCITYFACTOR** divided by 1000 (to convert milliseconds to seconds). Divide by 60000 to set acceleration units to velocity units per minute:

```
A1.AccelerationFactor = A1.VelocityFactor/1000 'Accel units = vel/sec
```

or

```
A1.AccelerationFactor = A1.VelocityFactor/60000 'Accel units = vel/min
```

## 3.6.10. Jerk Units

It is also necessary to define the jerk factor, even if you always use the smooth factor. The smooth factor automatically defines the jerk value from the velocity and acceleration values, but it is a value before factors, therefore totally invalid values of jerk (internally) can be computed. At least set Jfac=Afac/1000 and it should work. <***axis***>**.JERKFACTOR** specifies the conversion factor between your jerk units and the internal units [counts per msec$^3$]. <***axis***>**.JERKFACTOR** must contain the conversion factor for both the position dimension and the time dimension.

# *3. 7 ACCELERATION PROFILE*

The MC provides smooth, controlled acceleration profiles to reduce vibration and wear on the machine, and to allow high acceleration rates. The SERVO**STAR** MC allows you to independently control acceleration and deceleration to further reduce vibration. For detailed information any listed commands, refer to the *SERVO**STAR**® MC Reference Manual*.

**ACCELERATION** and **DECELERATION** control the acceleration rates. The following lines of code entered at the terminal window (or in your program) change the acceleration rates for Axis A1:

```
A1.Acceleration = 1000
A1.Deceleration = 1000
```

Do not enter these commands yet. Acceleration units should be set before these values are entered. You can impose limits which constrain the time during which acceleration and deceleration occurs. The motion includes three major parts: acceleration phase, constant velocity cruise phase, and deceleration phase.

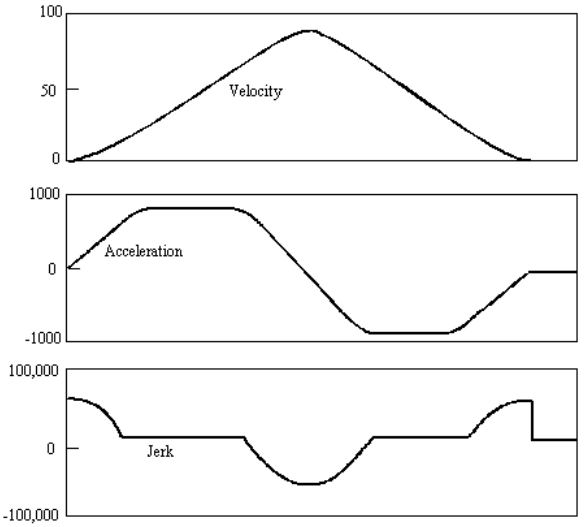
Time-based Acceleration/Deceleration is the motion interface in which the duration of the acceleration/deceleration phases during the movement is defined. These duration times will be used internally to calculate proper motion parameters to meet the time requirements as accurate as possible.

<***axis/group***>**.TIMEACCELERATION** and
<***axis/group***>**.TIMEDECELERATION** can be used for time-based definition.

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 which implies a very high level of jerk. The MC allows you to limit the amount of jerk in all profiles by using the axis properties **SMOOTHFACTOR** or **JERKFACTOR**.

Specify trapezoidal profiles by setting **SMOOTHFACTOR** (**SMOOTH**) to zero. If you want a smoother acceleration, 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.

Using <***axis***>**.ACCELERATION** and <***axis***>**.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 MC acceleration profile showing how controlled jerk produces smooth acceleration.




## 3. 8 POSITION AND VELOCITY

Position and velocity are the key command and feedback signals for each axis. These properties are updated by the MC every SERCOS cycle and may be read at any time. Their properties are double-precision floating point numbers. Velocity Units are discussed above. The four properties which hold these values are:

**<*axis*>.POSITIONCOMMAND** or **<*axis*>.PCMD**
**<*axis*>.POSITIONFEEDBACK** or **<*axis*>.PFB**
**<*axis*>.VELOCITYCOMMAND** or **<*axis*>.VCMD**
**<*axis*>.VELOCITYFEEDBACK** or **<*axis*>.VFB**

You can read these properties anytime the SERCOS ring is up.

Position error is defined as the difference between position commanded and position feedback. When an axis is at rest, (i.e., **ISSETTLED**), this simple definition of position error is valid. However, when the axis is in motion, the true position error does not equal the instaneous commanded position minus the instantaneous feedback position because there is a time delay of two SERCOS cycle times (2 ms) from when **MOVE** is issued until the feedback position is received.

When the MC calculates position error, it automatically accounts for the communication delay. In most circumstances, the two-cycle time delay is correct. However, if microinterpolation is enabled in the drive, the feedback position is delayed an additional one or more cycle times.

To manage position error problems, the MC provides a number of axis and group properties relating to position error. **<*axis*>.POSITIONERRORDELAY** (new in firmware version 3.0), allows you to specify the number of SERCOS cycle times to apply when calculating position error. The default is two.

The MC allows you to set up any axis in your system as a rotary axis. Rotary axes are 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 axis. In this case, the table units are set up as degrees and the units of the axis are set to repeat after 360°. 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.

Consider this 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:

```
A1.PositionFactor = 40000/1080
A1.PositionRollover = 360

A1.PositionRolloverEn = On‘Enable Rotary Motion
```

In this case, the feedback position is always between 0 and 360. 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.

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 MC. This is because inaccuracy is accumulated over many revolutions of the motor. For example, we could have rounded A1.POSITIONFACTOR in the above example to 37.037, which is accurate to 1 part in 1,000,000. However, after 10,000 revolutions (2000 rpm for just 5 minutes), that 1 part in a 1,000,000 would have accumulated to a few degrees. This means that if the table position is 100 degrees, it might read as 97°. In other words, the table appears to drift.

In the example above, the math is exact until the MC performs the division. Because the MC calculates with double precision (about 14 places of accuracy), you should use MC math when calculating ratios that cannot be represented precisely in decimal notation. In the example, using the full accuracy of the MC (about one part in 1014) the motor would have to travel about 3x1011 revolutions to accumulate one degree of error. That is equivalent to 88 years of continuous rotation at 6000 rpm.

All SERVO**STAR** drives provide interface hardware to accept an external or auxiliary encoder. This encoder is in addition to the primary position feedback device. You will need to set up the SERCOS telegram to support this. External position is frequently used in master-slave operations (gearing and camming) where the system is slaved to an external signal such as a line-operated motor.

The MC provides the external position through the axis property, **POSITIONEXTERNAL** or **PEXT**. This variable contains the accumulated encoder movement of the drive's external encoder input. It is updated every SERCOS cycle.

You can access **PEXT** two ways: realtime or on a as-needed basis. To access **PEXT** on an as-needed based, issue **IDNVALUE**. For example:

```
Dim Shared StorePExt as Double
StorePExt = IDNValue(1, 53, 7)
```

This gets **PEXT** in counts, not user units. You can access **PEXT** in realtime by properly configuring the telegram for the axis with the external encoder.

The SERVO**STAR** MC allows you to control the units of the **POSITIONEXTERNAL** (**PEXT**). This is controlled with <***axis***>.**POSITIONEXTERNALFACTOR** or <***axis***>.**PEXTFAC**. The process is identical to setting position units for an encoder. The external position and velocity factors are only effective for cases when the SERCOS telegram is configured to transmit the external encoder.

Do not access **PEXT** using **IDNVALUE** because the encoder values coming into the drive are subject to rollover. The MC continuously monitors these values and adjusts the value when a rollover is detected. The values accessed via the SERCOS service channel cannot be monitored for rollover. Also, external position units are not in effect.

The MC provides the external velocity through the <***axis***>.**VELOCITYEXTERNAL** or <***axis***>.**VEXT**. This variable contains the accumulated encoder movement per millisecond and is also updated every SERCOS cycle when the SERCOS telegram is configured to transmit the external encoder.

The MC allows you to control the units of **VELOCITYEXTERNAL** with <***axis***>.**VELOCITYEXTERNALFACTOR** or <***axis***>.**VEXTFAC**. The process is identical to setting position units except that <***axis***>.**VEXTFAC** can only be input as an encoder signal.

## *3. 9 LIMITS*

This section outlines the many types of limits you can impose on the MC system. These limits help protect the machine from excessive excursions of travel, speed, and torque. There are three types of limits in the SERVO**STAR** system:

MC Generator Limits
MC Realtime Limits
Drive Limits

Limits can be imposed in several ways. First, some limits are imposed by the MC and others by the SERVO**STAR** drive. Limits can be checked in realtime or they can be checked only for subsequent actions.

**MC 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 or acting as a slave, then **POSITIONMIN** and **POSITIONMAX** (position limits) are checked only at the beginning of the move.

The MC checks **realtime limits** each SERCOS update cycle. Changes in these limits affect current operations. **VELOCITYFEEDBACK** is checked against **VELOCITYOVERSPEED** in every SERCOS cycle.

**Drive limits** are imposed by the SERVO**STAR** CD drive. Changes in these limits affect current operations. For example, ILIM limits the peak current that is output by the drive. These limits are imposed independently of the position and velocity commands issued by the controller.

## 3.9.1. Position Limits

There are several limits in the MC related to position:

**POSITIONMIN** or **PMIN**
**POSITIONMAX** or **PMAX**

**POSITIONMIN** (**PMIN**) and **POSITIONMAX** (**PMAX**) place minimum and maximum limits on the position feedback for an axis. Set the limits with:

```
A1.PositionMin = -10
A1.PositionMax = 200.5
```

or you can use the short form:

```
A1.PMin = -10
A1.PMax = 200.5
```

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

Position limits are checked two ways. First, they are checked in the motion generator. If you command a position move beyond the position limits, an error is generated and the move does not start. Second, they are checked each SERCOS update cycle (generally, 2 or 4 ms). If the axis is moving and crosses the position limit, this generates an error.

If the axis has already stopped because the position limits were crossed, the MC does allow you to back out. In other words, you can enter commands that move the axis back within the limits. The axis will move without generating errors.

Position limits are continuously checked only when the drive is in master-slave mode. Some conditions may occasionally result in the motor moving outside **PMIN** or **PMAX** limits which are not detected by the MC. SERCOS loop instability may also cause position errors.

You can control whether the MC watches either one or both of the limits in position limit. To do this, set **POSITIONMINENABLE** and **POSITIONMAXENABLE** to either ON or OFF.

```
A1.PositionMinEnable = ON
A1.PositionMaxEnable = ON
```

or, you can use the short forms:

```
A1.PMinEn = ON
A1.PMaxEn = ON
```

The default state of **PMINEN** and **PMAXEN** is off. You must enable them to use these limits.

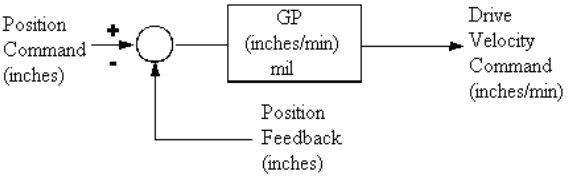
You should limit the maximum position error your system tolerates. You do this by setting <***axis***>**.POSITIONERRORMAX** (<***axis***>**.PEMAX**).

Care should be taken to set **PEMAX** to a value that matches the needs of the application. When the actual position following error (**PE**) exceeds **PEMAX**, motion stops. If the motion is stopped when this condition is detected, the axis is disabled.

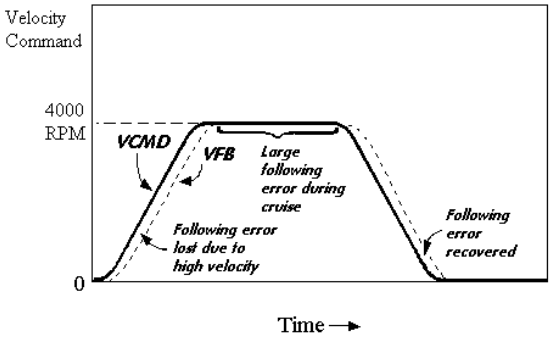
During normal operation, occasional occurrences of position error overflow usually indicates a malfunction of the machine, such as a worn or broken part, or a need for lubrication. You should set the maximum position error well outside the boundaries of normal machine operation or nuisance errors occur when the machine is running.

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 consequently, the motor is able to run away.

The proportional position loop is the simplest position loop. The next figure shows a block diagram of a proportional position loop:




As you can see, the velocity command is proportional to the following error. Large velocity commands need large following error limits. At first, the units of inches/min/mil may seem confusing. The following error for an acceleration with a typical proportional loop is shown in the next figure.




The good news about 100% feed-forward is that it eliminates steady-state following error. The bad news 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 is acceptable.

In many cases, 100% feed-forward is not acceptable. In these cases, you can reduce the feed-forward to reduce overshoot. The larger the feed-forward gain, the greater reduction is seen steady-state following error. Most systems can tolerate the overshoot generated by feed-forward gains of 50%.

**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 this technique is sometimes called inertial feed-forward.

Acceleration feed-forward works only when the inertia load is proportional to the axis acceleration. It does not work with axes that are coupled, such as 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, use **MOTIONLINK**, or set the acceleration feed-forward scaling constant from the MC. Acceleration feedforward is IDN 348. For an axis A1, the correct line of code is:

```
WriteIDNValue Drive = A1.DriveAddress IDN=348 Value = 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) gives good results. Fine-tune this value by applying trapezoidal trajectory and find the user gain that yields the minimum following error during the acceleration and deceleration.

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

```
A1.PositionErrorSettle = 0.01
```

or you can use the short form:

```
A1.PESettle = 0.01
```

to specify that the position error must be below 0.01 units 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 MC does this automatically through **STARTTYPE** when executing **MOVE** commands as is discussed in the Single Axis Motion section of this manual. You must set TSETTLE>0 or **POSITIONERRORSETTLE** is ignored.

**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 MC waits **TIMESETTLE** milliseconds for settling. If the position error goes above **POSITIONERRORSETTLE** during that time, the timer resets.

**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** is a property that indicates the state of the motion profiler. The valid range of values is from –1 to 3, with the following meaning:

-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 a1.ismoving > 0          'wait for profiler to finish
End While
```

## 3.9.2. Axis Velocity Limits

There are several limits in the MC related to velocity:

VelocityOverspeed
VelocityMax
Acceleration
Deceleration

**VELOCITYOVERSPEED** defines an absolute motor velocity limit. When this limit is exceeded, an error is generated and the axis is brought to an immediate stop. **VELOCITYOVERSPEED** is in MC velocity units and is set any time. It is checked every SERCOS update cycle. For example:

```
A1.VelocityOverspeed = 4000
```

**VELOCITYMAX** (**VMAX**) sets the maximum speed that the motion generator can command. **VELOCITYMAX** is in MC velocity units and is only set from the terminal window or in the configuration file, Config.Prg. It is checked at the beginning of each move. For example:

```
A1.VelocityMax = 5000
```

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

**ACCELERATIONMAX** is the upper limit for acceleration in motion commands. It is in MC acceleration units. **ACCELERATIONMAX** is set up in the BASIC Moves auto-setup program when you start a new project.

**DECELERATIONMAX** is the upper limit for deceleration in motion commands. It is in MC acceleration units. **DECELERATIONMAX** is set up in the BASIC Moves auto-setup program when you start a new project.

# *3. 10 VELOCITY, ACCELERATION AND JERK RATE PARAMETERS*

**Axis.VelocityRate** defines the axis velocity maximum scaling factor from 0.1 to 100 persents independently of acceleration, deceleration or jerk. <***axis***>**.VelocityRate** may be modal or nodal.

**Axis.AccelerationRate** defines the axis acceleration maximum scaling factor from 0.1 to 100 persent independently of velocity, deceleration or jerk. <***axis***>**.AccelerationRate** may be modal or nodal.

**Axis.DecelerationRate** defines the axis deceleration maximum scaling factor from 0.1 to 100 persent independently of velocity, acceleration or jerk. <***axis***>.**DecelerationRate** may be modal or nodal.

**Axis.JerkRate** defines the axis Jerk maximum scaling factor from 0.1 to 100 persents independently of velocity, acceleration or deceleration. <***axis***>.**JerkRate** may be modal or nodal.

For example:

```
A1.VRate = 50          ' VelocityRate  50%
A1.ARate = 70          ' AccelerationRate 70%
A1.DRate = 80          ' AccelerationRate 80%
A1.JRate = 60          ' JerkRate 60%
```

## *3. 11 VERIFY SETTINGS*

Now, you can check the system using the MC **MOVE** command. This command is discussed in detail later. For right now, use it to verify some of the settings above. First, enable your drive.

### 3.11.1. Enabling

Enable the drive with the following commands:

```
System.Enable = ON
A1.Enable = ON
```

After the SERCOS phase is set to 4, the drive can be enabled. This condition is indicated with an “S” on the front of each SERVO**STAR** amplifier. You do this with MC command:

```
Sercos.Phase = 4.
```

The code to do this is normally generated in the BASIC Moves auto-setup program which runs each time you start a new project. A decimal point at the bottom-left of the S on the SERVO**STAR** drive display should come on when the drive is enabled. The decimal point may flash under certain conditions.

You must turn on **SYSTEM.ENABLE** (**SYS.EN**) before turning on any of the drive-enable properties. This is because when **SYSTEM.ENABLE** is off, it forces all the axis-enable properties off.

### 3.11.2. Motion Flags

The next step in preparing a system for motion is turning on the motion flags. There are two motion flags that must be on: the system motion flag and the axis motion flag:

```
System.Motion = ON
A1.Motion = ON
```

### 3.11.3. Move

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

```
Move A1 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) and verify your position units:

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

# 3. 12 SERCOS

The MC uses the SERCOS fiber-optic ring network to communicate with the drives. The SERCOS interface (**SE**rial **R**ealtime **CO**mmunication **S**ystem) is an internationally recognized specification (IEC 1491), supported by many companies around the world. SERCOS replaces analog connections with digital, high-speed fiber-optic communication. Danaher Motion supports 2/4/8/16 Mbit/s baud rate speeds. Danaher Motion selected the SERCOS interface because of numerous technical advantages:

**Determinism**
SERCOS provides deterministic communication to all drives. The communication method provides complete synchronization between controller and axes, and among all axes.

**International Standard**
SERCOS is the only digital communication method for motion control supported by an international body. It is the only standard with motion control products provided by a wide base of companies.

**Reduced Wiring Between Drive And Controller**
SERCOS greatly reduces wiring between controller and drives. While analog and PWM amplifiers require 10 to 15 connections-per-drive, SERCOS requires only two fiber-optic cables between the controller and the drive.

**Noise**
Because SERCOS is based on fiber optics, electromagnetic noise is greatly reduced. The controller is electrically isolated from drives and feedback devices, as well as from limit switches and other I/O wired through the drives. Most grounding and shielding problems, notorious for the difficulties they cause during installation of analog motion control systems, are eliminated.

**Simplifying Replacement**
SERCOS allows you to download configuration parameters for all drives simultaneously via the SERCOS network. You can configure your system to automatically configure the drives after each power up. If a drive needs to be replaced, all re-configuration is automatically done.

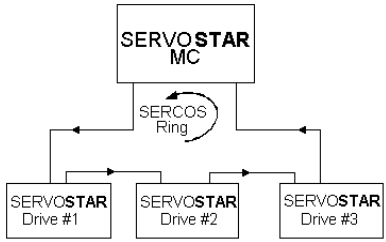
**Reliable Connections**
SERCOS uses fiber optic connections to eliminate the tens or even hundreds of hard-wired connections between the drives and the controller. This eliminates many possibilities for intermittent or reversed connections.

**Access to Drive**
SERCOS provides complete access to drive data. For the SERVO**STAR** drive, you can even record variables realtime in the drive and send the data via SERCOS to your PC using **MOTIONLINK**.

**Axis Modularity**
SERCOS relies on the drive to connect to most external signals related to an individual drive: feedback device, limit switches, and registration sensors. When an axis is added to a system, most of the wiring associated with the axis connects directly to the drive. This minimizes the changes to system wiring required to support adding or removing drives.

## 3.12.1. Communication Phases

In SERCOS, one master controller and many drives are connected with the fiber optic cable to form a communication ring.




The master attempts to establish communication in a step-by-step process. This process is referred to as bringing up the ring. Each step is defined in SERCOS as a phase. In order to bring up the ring, the system must proceed successfully through five phases (0 through 4). The MC simplifies this process by allowing you to specify at which phase you want the ring. In most cases, you simply need to set the SERCOS property phase to 4. The main exception to this is when configuring telegram Type 7 to include external encoder data, which requires that you first set the phase to 2 and then to 4.

In phase 0, the master establishes synchronization by passing the **M**aster **S**ynchronization **T**elegram (MST) to the first drive. That drive passes it to the next and so on until it is returned to the master, indicating that the ring is complete. After the master determines that 10 MST telegrams have been passed through the ring, phase 0 is complete.

In **phase 1**, the master sends the **M**aster **D**ata **T**elegram (MDT). Each drive is addressed individually and each drive responds with an **A**mplifier **T**elegram (AT). When all drives have responded, phase 1 is complete.

In **phase 2**, all drives are configured through a series of IDNs. First, the communication parameters are sent. Then, the drives are configured for operation.

Up to this point, all data have been sent via the service channel. In **phase 3**, the cyclic data channel is established. Configuration is completed and verified.

**Phase 4** is normal SERCOS operation.

SERCOS.PHASE automatically moves the SERCOS ring through each phase. You can observe this operation by watching the 7-Segment LED display on the SERVO**STAR** drive. The number displayed indicates the current communication phase. When the process is complete, an S (or 5) is displayed, indicating that the ring is up and drives can be operated.

## 3.12.2. Telegrams

All SERCOS data are organized into data packets called telegrams. Telegrams transport data and also provide protocol support and error checking. SERCOS provides three telegrams, which are issued in order:

**M**aster **S**ynchronization **T**elegram (MST)
**A**mplifier **T**elegram (AT)
**M**aster **D**ata **T**elegram (MDT)

The first telegram in a communication cycle is the **M**aster **S**ynchronization **T**elegram (MST). The MST, issued by the controller. It provides a single mark in time for all the drives. All feedback and command data are synchronized for all axes via the MST.

Immediately after the MST is issued, each drive returns feedback and status data in the **A**mplifier **T**elegram (AT). This includes position and velocity feedback as well as responses to non-cyclic commands.

The final telegram in the communication cycle is the **M**aster **D**ata **T**elegram (MDT). The controller issues the MDT. It contains information transmitted from the MC, such as signals and position and velocity commands. The service channel sends signals as well as non-cyclic commands.

## 3.12.3. Telegram Types

SERCOS provides for a variety of telegram types. Different telegram types define different sets of data to be exchanged in the AT and MDT. There are seven standard telegrams in SERCOS. The simplest telegram is Type 0, which defines only the service channel. No data are transferred in the cyclic data. The most complex type is Telegram Type 7, which allows the user to configure which data are in the cyclic data (all telegrams equally support the service channel.) The other telegrams (Types 1 through 6) define different combinations of position, velocity and current in the AT and MDT. The uses only Telegram Types 5 and 7 for data communication.

Normally, the relies on the SERCOS **telegram type 5**. In Telegram Type 5, position and velocity feedback are transmitted by the drive in the AT cyclic data, and position and velocity command are transmitted by the MC in the MDT cyclic data. These four components are the only motion data transmitted in the cyclic data. Telegram type 5 works well unless you need to have the external encoder position brought in with the cyclic data. This is necessary when you want to slave an axis to the external encoder In this case, you need to configure a Type 7 telegram.

When you need to bring an external encoder position from a drive to the MC in realtime (cyclic data) you must configure a **telegram type 7** for the drive. This telegram must contain all the motion data found in telegram type 5 (position and velocity in both the AT and MDT), with the external encoder position added.

To set up an axis for telegram type 7, you must modify the SERCOSSETUP subroutine, which is generated as part of the BASIC Moves auto setup program. You need to take the following steps:

- Set **SERCOS.PHASE** to 0 and set the baud rate
- Set the axis **PEXTFAC** of the axis with external encoder
- Set the drive addresses (**DRIVEADDRESS**)
- Set the axis **MASTERSOURCE**, **GEARRATIO**, and slave properties of the slave axis
- Set Sercos.Phase to 2
- Configure the axis for Telegram Type 7 using IDN 15
- Configure the AT for **VFB**, **PFB**, and **PEXT** (IDNs 40, 51, 53)
- Configure the MDT for **VCMD**, **PCMD** (IDNs 36, 47)
- Set **SERCOS.PHASE** to 4

See SERCOS Setup Subroutine in Appendix A for a sample subroutine.

## 3.12.4. Cyclic vs. Non-Cyclic Communication

SERCOS communicates across the fiber-optic ring with two types of data: cyclic and non-cyclic. Cyclic data are sometimes referred to as real-time, and are updated at a regular rate (SERCOS update rate). The SERVO**STAR** system supports update rates from 1 millisecond. The MC requires that the cyclic data include position and velocity command (transmitted from the controller) and position and velocity feedback (transmitted from the drives.) These data must be updated once each SERCOS update.

Non-cyclic data are updated irregularly and on a demand basis via the service channel. The service channel is roughly equivalent to a 9600-baud serial link between the drives and the controller, except that it is transferred on the fiber optic cable along side the cyclic data. Using the service channel, you can request data from the drive or set a parameter that resides in the drive. Each SERVO**STAR** drive has I/O points. You can use the service channel to access this I/O or you can use a Telegram type 7 to configure the I/O as cyclic data. The service channel is non-deterministic. Consequently, non-cyclic data transmits considerably slower than cyclic data.

## 3.12.5. IDNs

SERCOS commands are organized according to an Identification Number or IDN. SERCOS defines hundreds of standard IDNs, which support configuration and operation. Manufacturers, such as Danaher Motion, provide IDNs specific to their products. IDNs are numbered. Standard IDNs are from 1 to 32767 (although only several hundred are used to date) and manufacturer IDNs are numbered from 32768 to 65535.

If you are using a SERVO**STAR** drive, you normally use only a few IDNs for special functions (e.g., for querying drive I/O or changing the peak current of a drive).

SERCOS requires that you define every IDN used on each drive. The SERVO**STAR** system eliminates most of this step because the MC automatically defines all necessary IDNs for SERVO**STAR** drives. This process would otherwise be tedious as there are quite a few IDNs that must be defined.

Most requests from the service channel are responded to immediately. For example, when you set the digital-to-analog converter (DAC) on the drive, the value is placed in the DAC by the drive immediately upon receipt. The drive acknowledges the command almost as soon as it is received, thereby freeing the service channel for subsequent commands. However, some functions of the drive that are accessed from the service channel require a much longer time for execution. For example, incremental encoder-based systems must be oriented on power up. The wait for verification that the process has completed successfully can take many SERCOS cycles, far too long to tie up the service channel. To support these functions, SERCOS developed procedures.

With procedures, the master starts a procedure and optionally halts and restarts it. Procedures allow processing in the service channel without blocking other communication. For example, waiting for a registration mark is a procedure. In this case, the motor is turning and the drive is searching for a registration mark. By using a procedure, the service channel remains available for other communication during the search.

## 3.12.6. Position and Velocity Commands

The MC sends position and velocity commands in the cyclic data, and SERVO**STAR** drives return position and velocity feedback in the cyclic data. This allows you to configure your system as a position or a velocity loop. It also allows you to switch between position and velocity loop on-the-fly. The format of each data type is shown below:

| Signal | Size | Scale |
|---|---|---|
| Position Command | 32 Bit | One count |
| Position Feedback | 32 Bit | One count |
| Velocity Command | 32 Bit | 1/256 count per SERCOS update |
| Velocity Feedback | 32 Bit | 1/256 count per SERCOS update |

## 3.12.7. SERCOS Support

The MC provides numerous commands within the language to support SERCOS.

**Sercos.Baudrate for SERCON 410B:**
The SERCOS baud rate is set to 2 (2MBits/sec) or 4 (4MBits/sec). The baud rates of all drives must be the same. To control the baud rate on the drive, find the 10-position DIP switch on top of the drive and set the switches according to the following:

| Baud rate | Sercos.Baudrate | Drive, Switch 6 |
|---|---|---|
| 2 MBits/sec | 2 | Off |
| 4 MBits/sec | 4 | On |

**Sercos.Baudrate for SERCON 816:**
The SERCOS baud rate is set to 2 (2 M bits/sec) / 4 (4 M bits/sec) / 8 (8 M bits/sec) or 16 (16 M bits/sec). The baud rates of all drives must be the same. To control the baud rate on the drive, find the 10-position DIP switch on top of the drive and set the switches according to the following:

| Baud rate | Sercos.Baudrate | Drive, Switch 6 | Drive, Switch 10 |
|---|---|---|---|
| 2 M bits/sec | 2 | Off | Off |
| 4 M bits/sec | 4 | On | Off |
| 8 M bits/sec | 8 | Off | On |
| 16 M bits/sec | 16 | On | On |

Remember that DIP Switch-6 and DIP Switch-10 are read by the drive only on power up. If you change DIP switch-6 or DIP Switch-10, you must cycle power on the drive. Some versions of drives exists SERCON816, but in compatible mode – in this case it equals to SERCON 410B. For checking drive sercon version and mode see drive product number.

For the current SERCON version, query from the terminal with:? System.SERCONversion. It returns:

→2 For SERCON 410B Version or
→16 For SERCON 816 Version

**Sercos.CycleTime** is used to either set or query the SERCOS communications cycle time - rate of updating cyclic data (rate with which to the desired phase of communication (see Cyclic vs. Non-Cyclic Communication). This property is only set in SERCOS communication phases 0, 1, and 2, transferred from the MC to drives during phase 2 and becomes active in phase 3.

```
sercos.cycletime =  2000
? sercos.cycletime
```

**Sercos.NumberAxes** is used for querying the actual number of SERCOS drives detected on the ring. This value is generated during communication phase 1, when the SERCOS driver scans the ring for drives (or when the the drive addresses are explicitly set)

```
? sercos.numberaxes
```

**Sercos.Power** is used to set or query the SERCOS transmission power. The transmission power is governed by the SERCON interface chip and is set to one of six levels. Setting the power level too high drives the fiber-optic receiver into saturation and causes communication errors. This value should not have to change unless a very long cable is being used (>10 meters). Level 6 is the highest power.

```
Sercos.power = 3
? sercos.power
```

**Sercos.Scan** indicates to the SERCOS driver whether to scan the ring for drives. If the property is set to a value greater than 0, the ring is scanned during communication phase 1. The scan value indicates the highest drive address for which to scan. For example, if the value is set to 5, the ring is scanned for drives at addresses up to 5.

```
Sercos.scan = 0         'Do not scan the ring
Sercos.scan = 5         'Scan for drives up to address 5
```

Set **SERCOS.PHASE** to the desired phase of communication (see Enabling and Communication Phases). Normally, the command stream outlined in the procedure below is sufficient to bring up the ring. Because it takes time to bring up the SERCOS ring, you may want to speed the process by changing the phase only when it is at some phase other than 4.

Most IDNs have numeric values. After these IDNs are defined, you can read their values with **IDNVALUE**. To do this, you must specify the drive address, IDN, and the IDN element. For example, you can enter:

```
? IDNValue{5, 104, 7}
```

This lists the value of IDN 104, element 7, for the drive at address 5. **IDNVALUE** is only executed in SERCOS communication phases 3 and 4 as cyclic data, or in phases 2, 3, or 4 via the service channel (some restrictions apply for writing to an IDN). For a complete listing of IDNs supported by the SERVO**STAR** drive, refer to the *SERVO**STAR**® MC Installation Manual*.

IDNs have numeric values and you can write to most of them with the **WRITEIDNVALUE** command. To do this, specify the drive address, IDN and the value. For example, you can enter:

```
WriteIDNValue Drive = 5 IDN = 104 value=1
```

which sets the position loop gain (IDN 104) of drive 5 to 1. **WRITEIDNVALUE** is executed only in SERCOS communication phases 2, 3 and 4.

Some IDNs have alpha-numeric string values. After these IDNs are defined, you can read the string values with **READIDNSTRING**. Specify the drive address, IDN, and IDN element. For example, you can enter:

```
? IDNString(5, 104, 7)
```

**IDNSTRING** is used only in SERCOS communication phases 2, 3 and 4.

Some IDNs have alpha-numeric string values. After these IDNs are defined, you can write new string values to most of them with **WRITEIDNSTRING**. Specify the drive address, IDN and the value. For example, you can enter:

```
WriteIDNString Drive = 5 IDN = 104 Element = 7
String = IDN String
```

**WRITEIDNSTRING** is executed only in SERCOS communication phases 2, 3 and 4.

**IDNState** returns the state of a specified procedure command from a specified drive. It is used when executing a procedure command to determine progress of that procedure's execution. Common procedures are 99 (clear faults) and 148 (drive-controlled homing).For example, you can enter:

```
'Check how procedure terminated
If IdnState(a1.dadd, 99) <> 3 Then
     Print "Reset Faults procedure failed"
End If
```

*For more information about states of this property, refer to the SERVO**STAR**® MC Reference Manual.*

Use **MOTIONLINK** to configure and operate the SERVO**STAR** drive. The parameters are stored in the drive, but not in the controller. So, if you need to replace a drive in the field, you must reconfigure that drive with **MOTIONLINK**.

One benefit of SERCOS is that it allows you to download all drive parameters via the SERCOS link each time you power up. If you need to replace a SERVO**STAR** drive, you do not need to configure it with **MOTIONLINK**. This is a significant advantage when maintenance people unfamiliar with the design of the system do drive replacement. For this reason, Danaher Motion recommends that you configure your systems to download all parameters on power up. BASIC Moves provides a utility that reads all pertinent IDNs after the unit is configured and the ring is up and stores them in a task you automatically download each power up.

## 3. 13 LOOPS

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

- Standard position loop
- Dual-feedback position loop
- Velocity loop

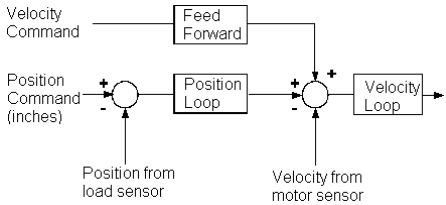
The MC sends position and velocity commands each SERCOS update cycle. The operation of the MC is hardly affected by the loop selection because the loop selection is made in the drive.

## 3.13.1. Standard Position Loop

Position loop is the standard operating mode of the SERVO**STAR** system. For position operation, set the position loop gain to a value greater than zero. Typically, you set up the position loop using **MOTIONLINK**. See the *SERVO**STAR**® MC Installation Manual* for more information.

## 3.13.2. Dual-feedback Position Loop

Dual-feedback position loop compensates for non-linearity in mechanical systems (inaccuracy from gears, belt drives, or lead screws, or backlash from gears and lead screws). Dual-feedback position loop adds a second feedback sensor (usually a linear encoder) to the system so the load position is directly determined. Unfortunately, the load sensor cannot be the sole feedback sensor in most systems. Relying wholly on the load sensor usually causes machine resonance that severely affects system response. Dual-feedback position loop forms a compromise: uses the load sensor for position feedback (accuracy is critical) and uses the motor sensor for velocity feedback (low compliance required). This is the dual-feedback position loop shown below.

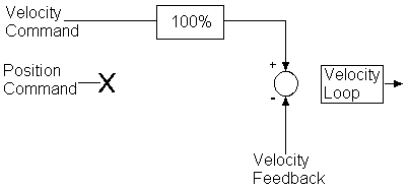



The additional hardware for dual-feedback position loop is a second encoder connected to the drive which is running dual-feedback position loop. You configure the drive for dual-feedback position loop operation using **MOTIONLINK**. See the *SERVO**STAR**® MC Installation Manual* for more information. The SERVO**STAR** MC does not close the servo loops and is not directly involved in dual-feedback position loop operation. In a sense, the MC is not aware that the drive is operating in dual-feedback position 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 MC to monitor the external encoder for position error. This is done with <***axis***>**.FEEDBACK**. For example:

```
A1.Feedback = External    'Set the axis feedback to the load
```

### 3.13.3. Velocity Loop

For some applications, the motor must be under velocity loop control. Maintaining a position is not important. The advantage is that the motor is more responsive (typically 3 to 5 times faster in settling time). When you configure an axis for velocity loop, you more or less fool the drive by setting gains to disable the position loop. You need to set the position loop gain to zero and the feed-forward gain to 100%. This produces the following servo loop:




Since a lot of position error can accumulate in velocity loop, it is best to configure the system to have such a large maximum position error that it never realistically generates an error.

## *3. 14 SIMULATED AXES*

A simulated axis is an axis that exists only inside the MC. Simulated axes have most of the functions of physical axes including profile generators, units, position and velocity commands and feedback signals, and limits. However, simulated axes do not include a drive, motor or feedback device.

An axis can only be configured as a simulated axis during Sercos.Phase 0 or before SERCOS configuration takes place in CONFIG.PRG or AUTOEXEC.PRG. To set up a simulated axis, set the axis property **SIMULATED** to ON:

```
A1.Simulated = ON
```

This step is inserted in the BASIC Moves auto setup program, in the subroutine, SercosSetup, just after the phase is set to zero. See SERCOS Setup Subroutine in Appendix A for additional details.

Simulated axes require about the same amount of resources from the MC, as physical axes. As a result, simulated axes are part of the total axis count of the controller. For example, a system with three physical axes and one simulated axis requires a four-axis MC. Simulated axes are used in the MC in the same way as physical axes. They are profiled with **JOG** and **MOVE**, can be geared and cammed, and can act as masters for geared and cammed axes. Variables from simulated axes such as position feedback can be used to generate events. During operation, the feedback position and velocity are set to their respective command values.

Although most commands and variables for axes work for simulated axes, there are some exceptions. Simulated axes cannot receive SERCOS commands. You cannot set drive variables such as position loop gains in simulated axes.

The units on a simulated axis are flexible. The simplest way to set the units is to set PFAC=1, which can be thought of as 1 simulated unit (eg, 1 meter, 1 inch). Set **VFAC** to PFAC/1000 for units-per-second or set to PFAC/60000 for units-per-minute such as rpm. Set **AFAC** to VFAC/1000, such as meters-per-second$^2$. Most axis properties do work with simulated axes, including:

- Position properties: command, feedback, final
- Velocity properties: command, feedback, cruise, maximum
- Acceleration and deceleration
- Acceleration maximum and deceleration maximum

In addition, you can move, jog, gear, and cam simulated axes. The following axis properties do not work with simulated axes:

- Position error and velocity error (assume to be zero)
- Velocity Overspeed
- Tuning parameters
- External position and velocity

# 4. SINGLE-AXIS MOTION

The SERVO**STAR** MC supports three main types of motion:

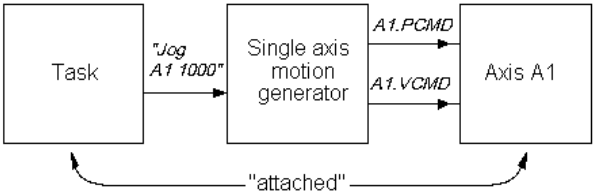
- Single-axis motion
- Master-Slave motion
- Multiple-axes motion

A motion generator controls all motion in the MC. This software device receives commands from MC tasks and produces position and velocity commands for the drives every servo cycle. The two main single-axis motion commands are **JOG** and **MOVE**. The MC also provides ways to synchronize the execution of motion commands as well as changing motion profiles on-the-fly. In addition, the MC provides multiple modes for starting and stopping motion.

All motion commands are subject to inherent system delays due to SERCOS communications. One cycle time is required to transmit a command. A second cycle time is required to receive the position feedback. Thus, the minimum system delay is 2 SERCOS cycles. If microinterpolation is enabled in the drive, there is an additional delay of 1 cycle time. Additional, indeterminate, delays may result from the servo system dependent on drive tuning and kinetics of the mechanical system.

## *4. 1 MOTION GENERATOR*

The basis of all motion in the MC 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 2 or 4 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.




### 4.1.1. Motion Conditions

Several conditions must be met before the MC generates motion:

- The controlling task must attach to the controlled axis.
- The controlled axis must be enabled.
- The system and axis motion flags must be enabled.

Before a task can send motion commands to an axis, **the axis must be attached to the task**. Attaching is also required for gearing. This prevents other tasks from attempting to control the axis. To attach an axis, issue **ATTACH** from the controlling task:

```
Program
Attach A1
```

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** can be 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:

```
Detach A1
```

If you issue **DETACH** while profile motion for that axis is in progress, **DETACH** delays execution until that motion is complete. If a task ends with some axes still attached, all axes are 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.

**Enable the drive** by typing the following commands in the terminal window:

```
System.Enable = ON
A1.Enable = ON
```

**Enable** is also necessary for asimulated axis although there is no drive connected.

There are two other signals that are required to enable the drive: **DRIVEON** and **HALTRESTART**. These signals are normally ON. These signals are provided to comply with SERCOS. They are seldom used in MC programs. If you do turn either of these off, you will need to turn them back on before motion is allowed.

The last step in preparing a system for motion is turning on the **motion flags**. There are two motion flags which must be on: the system motion flag and the axis motion flag. For example:

```
System.Motion = ON
A1.Motion = ON
```

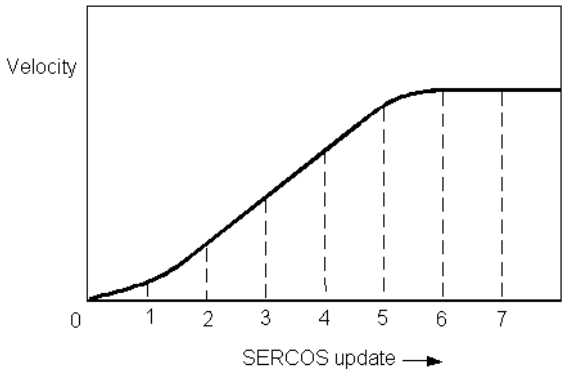
These lines of code are included in the BASIC Moves auto setup program.

In many applications, a **hardware switch** is tied to the motion flag. The SERVO**STAR** MC does not have a hardware motion input. However, you can use MC events to create this function. The following example shows a task that uses input external data input 1 to control the system motion flag:

```
Program
   OnEvent MOTION_ON System.DIn.1 = ON
      System.Motion = ON
   End OnEvent

   OnEvent MOTION_OFF System.DIn.1 = OFF
      System.Motion = OFF
   End OnEvent
End Program
```

The purpose of the motion generators 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 via SERCOS to the controlled SERVO**STAR** drive. All the servo loops (position, velocity, and current) are closed in the drives. The next figure shows the profile for the velocity portion of a **JOG** command.




The profile must be updated at regular intervals. The MC usually updates the profile every cycle time (1 ms **update rate**, if possible when the drive supports this update rate). The SERCOS telegram also gets longer with more axes. When using the MC with Pentium™ CPU board, the recommended update time to controls 8 or fewer axes is 2 ms and 4 ms when the MC controls 9-16 axes. When the MC is used with Pentium™III CPU, it controls up to 32 axes at 2 ms update rate.

## 4.1.2. Motion Buffering

The motion generator for each axis can process only one motion command at a time because the axis can process only one position or velocity command. However, you can send a second motion command to the generator where it is held until the current motion command is complete (motion buffering).

Motion buffering is controlled with **STARTTYPE**. If **STARTTYPE** is set to GENERATORCOMPLETED, the buffered profile starts immediately after the current profile is complete. If **STARTTYPE** is set to INPOSITION, 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.

You can also 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, set **STARTTYPE** to IMMEDIATE (IMMED) or SUPERIMMEDIATE (SIMM).

## 4.1.3. Override versus Permanent

Axes have numerous properties such as acceleration and deceleration. These properties are normally set directly:

```
ConveyorAxis.Acc = 100
```

This setting is permanent. It persists until the next time the property is assigned a value. For convenience, the MC 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:

```
ConveyorAxis.Acc = 100
Jog ConveyorAxis 1000 ACC = 10
```

Even though the acceleration rate of the conveyor axis is specified at 100 in the first line, **JOG** accelerates at 10 (the override value). 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 below as well as in the *SERVO**STAR**® MC Reference Manual*.

## 4.1.4. Acceleration Profile

The single-axis motion commands, **MOVE** and **JOG**, produce acceleration curves that are limited by:

**ACCELERATION** (**ACC**)
**DECELERATION** (**DEC**)
**SMOOTHFACTOR** (**SMOOTH**)

Limit **ACCELERATION** and **DECELERATION** to less than the acceleration capabilities of the motor and load. **SMOOTH** helps limit or reduce mechanical wear on the machine by smoothing motion.

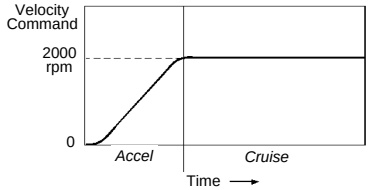
There are two types of acceleration profiles – sinus-curve profile and trapezoidal profile. In the sinus profile the acceleration is smoothly increased while in the trapezoidal profile, the acceleration is increases in one sample. Setting the acceleration increasing smoothness is done using **SMOOTHFACTOR**.

## 4.1.5. Jog

When you want to command the motor to move at a constant velocity independent of the current position, use **JOG**. Velocity can be negative to produce motion in the reverse direction. For example:

```
Jog ConveyorAxis 2000
```

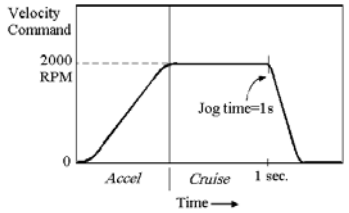
produces this profile:




You can optionally limit the amount of time **JOG** runs using **TIMEJOG**. **TIMEJOG** must be placed on the same line with **JOG**. **TIMEJOG** is specified in milliseconds and includes acceleration time. Deceleration starts when the time expires. When **TIMEJOG** is -1, **JOG** continues indefinitely. **TIMEJOG** defaults to -1. For example:

```
Jog ConveyorAxis 2000 TimeJog = 1000
```

produces the profile below.




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

**TIMEJOG**
**ACCELERATION**
**DECELERATION**
**SMOOTHFACTOR**
**JERK**
**VELOCITYRATE**
**ACCELERATIONRATE**
**DECELERATIONRATE**
**JERKRATE**

## 4.1.6. Stop

**STOP** stops motion in the motion buffer. In the command, you must specify the axis. For example:

```
Stop ConveyorAxis
```

Normally, **STOP** is set to stop motion immediately at the rate of **DECELERATIONMAX**. However, you can modify the effects of **STOP** with **STOPTYPE**. **STOPTYPE** can take three values:

***StopType = Immediate* or *IMMED***
Stop axis immediately at **DECELERATIONMAX** rate

***StopType = EndMotion***
Stop axis at end of current motion command and clear buffered motion

***StopType = Abort***
Stop the current motion immediately without ability to restore the stopped movements. Only the current motion command is stopped, the commands coming after are generated.

**STOPTYPE** defaults to IMMED. If **STOPTYPE** = ENDMOTION, the current move is completed before **STOP** is executed. For **JOG** with **TIMEJOG** = -1 (continue indefinitely), **STOPTYPE** = ENDMOTION has no meaning because **JOG** never ends. During **STOP**, the current and pending movements in the buffer are stored to allow recovery of these movement using **PROCEED**.

**STOP** uses the modal maximum deceleration and maximum jerk. **STOPTYPE** is overridden as part of a **STOP**.

## 4.1.7. Proceed

When a task stops motion in axes 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 MC supports **PROCEED**.

**PROCEED** 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**. The path control is provided by <***axis***>**.PROCEEDTYPE**. **PROCEEDTYPE** can be set in one of three modes: CONTINUE, NEXT MOTION, and CLEARMOTION. The key rules of operation are:

1. When a task stops the motion of axes to which it is attached, a **PROCEED** is allowed, but not required.
2. When **STOP** 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**. The task and the motion it commands can restart only after a **PROCEED** is issued from the terminal window. Motion is commanded from the terminal window before **PROCEED** is issued.
3. When **STOP** 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**. The task and the motion it commands restart only after **PROCEED** is issued from the task that stopped motion. Motion *cannot* be commanded from the task that stopped motion.
4. When **STOP** is issued from the terminal window for a command given from the terminal window, the motion is stopped and cannot proceed. In this case, the only available proceed type is CLEARMOTION from the terminal.

There are three ways for **PROCEED** 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 followed by the pending motion in the stopped buffer.

***ProceedType = NextMotion***
This causes the motion generator to abort the current move and go directly to the move in the stopped motion buffer.

***ProceedType = ClearMotion***
This clears the motion buffer. All motion commands in the stopped motionbuffer are aborted. This is the default.

## 4.1.8. Move

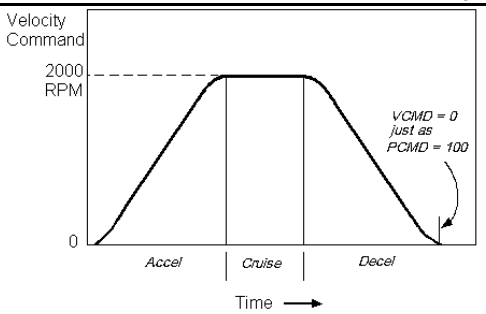
**MOVE** is the most common of point-to-point moves. The basic move is a three-segment motion:

| | |
|---|---|
| Accelerate | Go from zero speed to VelocityCruise |
| Cruise | Continue at VelocityCruise. |
| Decelerate | Go from VelocityCruise to VelocityFinal |

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** = 0) just as the position command reaches the final position. For example, the following command:

```
Move ConveyorAxis 100 VCruise = 2000
```

produces the next profile.




## 4.1.8.1. Position Final and Incremental Versus Absolute Moves

**POSITIONFINAL** (the end of a move), is always specified in **MOVE**. The meaning of **PFINAL** depends on **ABSOLUTE** (**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 is equal to **PFINAL**.

**Incremental (*Absolute = FALSE)***
Final position is referenced to the start position. The final position is equal to the sum of **PFINAL** and **PCMD** from the start of the move. So:

```
CutAxis.Absolute = TRUE
Move CutAxis 100
```

moves CutAxis to position 100. On the other hand:

```
CutAxis.Absolute = FALSE
Move CutAxis 100
```

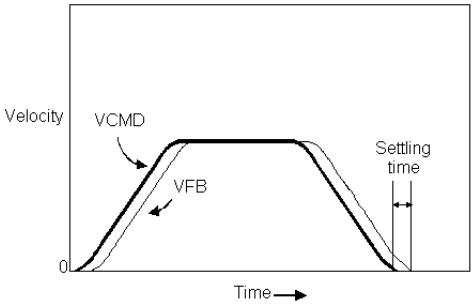
moves CutAxis a distance of 100 units from the current position.

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

## 4.1.8.2. Settling Time

The MC actively watches to see if axes are 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.

Consider the point-to-point move shown below. The **VELOCITYCOMMAND** (**VCMD**) is shown in solid line and **VELOCITYFEEDBACK** (**VFB**) is shown in thin line. 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.




Ideally, settling time allows the motor to move to approach zero position error. However, you must allow for the condition where the position error never quite reaches zero. On the MC, you specify how low you consider to be low enough with <***axis***>.**POSITIONERRORSETTLE** (<***axis***>.**PESETTLE**):

```
CutAxis.PESettle = 0.01
```

After the motion generator has completed a move which ends 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 MC allows you to specify this time period with <***axis***>.**TIMESETTLE** (given 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. If position error is within the **PESETTLE** range for the time specified by **TSETTLE**, the <***axis***>.**ISSETTLED** flag is TRUE (1). Otherwise, it is FALSE(0).

## 4.1.8.3. *Point-to-Point Moves*

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

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

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

A 1 second delay is forced 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 **SLEEP**.

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

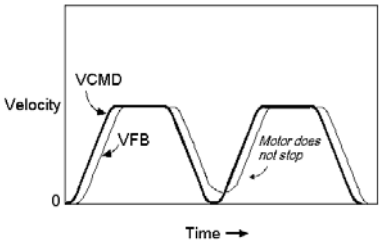
***StartType = GeneratorCompleted*** **(GCom)**
When STARTTYPE = GCOM, the new move starts as soon as the motion generator has completed the motion for the current move. GENERATORCOMPLETED is constant equal to 3. This command incurs a system delay of 2 SERCOS cycle times.

***StartType = InPosition (InPos)***
When STARTTYPE = INPOS, the motion generator delays executing the new motion command until the current move has completed and the position error is settled to near zero. **INPOSITION** is constant equal to 2. This command is not subject to additional system delays, 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. Use this 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 is constant equal to 1. This command incurs a system delay of 5 SERCOS cycle times each execution. This time is required to blend the previous move with the new move.

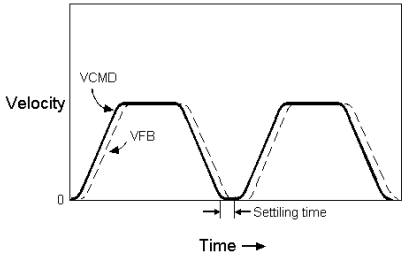
***StartType = SuperImmediate (Simm)***
SUPERIMMEDIATE is a variation of IMMEDIATE. The main difference is that SUPERIMMEDIATE eliminates the 5 SERCOS cycles delay by doing pre-calculation online, rather than offline in the motion manager. SUPERIMMEDIATE is constant equal to 5. The numbers of SUPERIMMEDIATE changes at a time in the system is limited by the load of the system and the type of move. SUPERIMMEDIATE is best used for short, high-speed movements.

When **chaining multiple moves** which 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. For example, the next figure shows this problem when **STARTTYPE** is incorrectly set to GCOM rather than INPOS.




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

Normally, the desired performance is for the velocity to 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. This is done by setting **STARTTYPE** to INPOS. This is shown in the following figure.




As you can see, the motor comes to rest because the MC waits for the following error to be small enough before proceeding to the next command.

```
MainAxis.PESettle = 0.01
MainAxis.Tsettle = 10
Move MainAxis 100
Move MainAxis 200 StartType=InPos
```

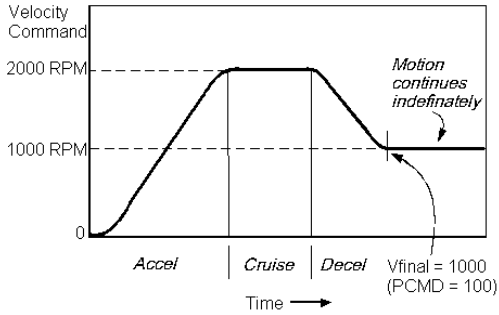
Normally, the ending speed of **MOVE** defaults to zero. However, you can specify an end speed other than zero (**Non-Zero End Moves)**. To do this, use <***axis***>**.VELOCITYFINAL**. **VELOCITYFINAL** must be specified on the same line as **MOVE**. For example,

```
A1.VelocityCruise = 2000
Move A1 100 VelocityFinal = 1000
```

or

```
Move A1 100 VelocityCruise = 2000 VelocityFinal = 1000
```

both produce the next profile.

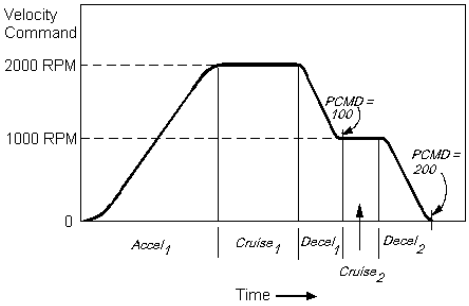



You seldom want to use a single move with a non-zero final velocity because motion continues until the specified ending position and abruptly begins decelerating to zero. The final position is not specified. Normally, moves with non-zero final velocities are used to build multi-step profiles.

Combining non-zero end-point moves in the motion buffer produces **multi-step moves**. For example:

```
Program
 Attach A1
 Call AxisSetup
 Call SercosSetup
 Sys.En = On
 A1.En = On
 Sys.Motion = On
 A1.Motion = On
 A1.Abs = Off
 A1.StartType = Gcom
 'Do Two-step Move
 Move A1 100 VCruise = 20 VFinal = 10
 Move A1 100 VCruise = 10 VFinal = 0
 'Wait for move to complete and axis to settle out
 While A1.IsSettled = 0
  Sleep 10
'Sleep in loop to keep from overloading CPU resources
 End While
 'Disable, detach and exit
 A1.En = Off
 Detach A1
End Program
```

produces the following profile.




In the above example, A1.STARTTYPE = GCOM. The second move begins 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 immediately 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 INPOS adds error if the second move is incremental. 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, there are a few restrictions:

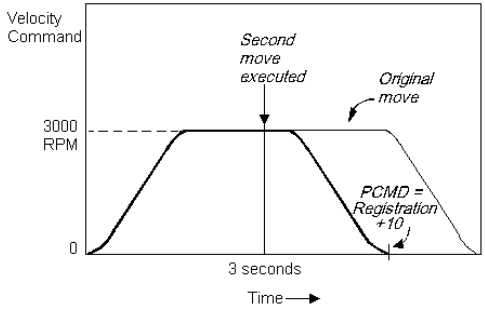
1. STARTTYPE for all non-zero end-point moves (VELOCITYFINAL<>0) must be GCOM.
2. **VELOCITYCRUISE** and **VELOCITYFINAL** are always positive. The direction is set by target position.

3. **PFINAL** of the succeeding move must be far enough in front of **PFINAL** of the current move 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.

You can change the end position or 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, change the end position without changing the velocity. For this, you would write:

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

This generates the profile shown below.




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

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

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

1. STARTTYPE must be IMMED or SIMM
2. Direction is set according to the target position. For Jog, according to the sign of the velocity.
3. New and old VELOCITYCRUISE and VELOCITYFINAL must be positive in case of move command.
4. If the succeeding move changes PFINAL or VFINAL, it must remain possible to create the profile with the axis acceleration limits

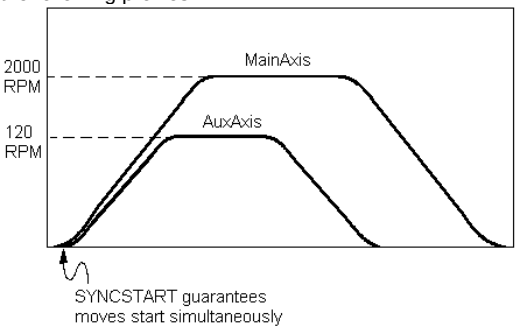
## 4.1.8.4. SYNCHRONIZING MULTIPLE AXES

The MC is abile to synchronize many single-axis **MOVE**s so they all start simultaneously. This is useful when you have multiple axes with moves that are largely independent, but must start at the same time. It is also useful for coordinating more than three axes. Group motion, which is covered later, provides tighter coordination, but with one movement profile.

Synchronization is controlled with STARTTYPE = SYNCSTART (SYNC). The feature works by allowing you to load the motion generator with a motion command, but delaying the generation of motion until a SYNCSTART command is issued. For example, the following sequence:

```
Move MainAxis 300 VCruise = 1200 StartType = Sync
Sleep 1000     'Program delayed between Move Commands
Move AuxAxis 100 VCruise = 1000 StartType = Sync
SyncStart MainAxis AuxAxis
```

generates the following profiles.




You can synchronize as many axes as you want, including simulated axes. Since each SYNCSTART specifies the axes it synchronizes, you can synchronize multiple sets of axes independently.

This command incurs a minimum system delay of 2 SERCOS cycle times, but the delay may be greater, depending on the number of moves synchronized together and the total load of the system.

If you have loaded a synchronized move into the motion generator and need to delete it, use **SYNCCLEAR**. For example, if you entered the following command before issuing **SYNCSTART**:

```
SyncClear MainAxis
```

the MainAxis move is deleted. This only affects subsequent moves. You can only use **SYNCCLEAR** to clear pending moves. **SYNCCLEAR** has no effect once the move is executing. Stopping the axis in this case is performed as for a non-synchronized move.

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

**ABSOLUTE**
**ACCELERATION**
**DECELERATION**
**SMOOTHFACTOR**
**VELOCITYCRUISE**
**VELOCITYMAX**
**STARTTYPE**
**VELOCITYRATE**
**ACCELERATIONRATE**
**DECELERATIONRATE**
**JERKRATE**

## 4.1.9. Velocity Override

The MC can speed up or slow down all motion commands. This can be applied to the entire machine at once, or to individual axes, independently. 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 the operation of the machine at a variety of speeds without modifying your program.

All axes on an MC are controlled with **SYSTEM.VELOCITYOVERRIDE** (**SYSTEM.VORD**). For example:

```
System.VelocityOverride = 25
```

immediately reduces the velocity commands of all currently executing **MOVE**s and **JOG**s and any subsequent commands to 25% of the current values. Since the velocities are controlled by **VORD**, the acceleration rates of all axes are proportionally adjusted. The final positions of **MOVE**s are not affected.

If you want to override the velocity of a single axis rather than the entire machine, you can use <***axis***>**.VORD.VELOCITYOVERRIDE** (<***axis***>**.VORD**). It is used in the same way as **SYSTEM.VORD** except it applies only to one axis. <***axis***>**.VORD** can be applied to as many or as few axes as desired and the amount of override specified for one axis is independent of the others. If you use **SYSTEM.VORD** and <***axis***>**.VORD** simultaneously, the axis speed is reduced by the product of both override properties as shown below:

```
System.VOrd = 66    'Reduce entire system to 2/3 speed
MainAxis.VOrd = 50  'Reduce MainAxis to 1/3 speed
```

In this example, the entire system is reduced to 66% and the MainAxis is reduced to 50% of the system speed which is 33%. There is a delay of 5 samples until the velocity start to change that to insure a smoothness velocity exchange.

# *4. 2 MOTION EXAMPLES*

This section provides a few examples that apply the motion control techniques discussed in this chapter to common machine functions.

## 4.2.1. Position Capture

Many drives (including all current SERVO**STAR** drives) implement the position capture functionality, commonly used in homing and registration procedures. Starting with version 3.0, the MC uses the SERCOS procedure mechanism to accomplish positioning.

While position capture is a drive-based function, the MC provides various commands to control the capture procedure. SERVO**STAR** drives provide three digital inputs (IN_1, IN_2, and IN_3), any one of which can be designated for the capture trigger signal. The command properties provided in the MC are:

<***axis***>**.Capture** Position capture is executed once the SERCOS relevant procedure (IDN 170) has been started. <***axis***>**.Capture** encapsulates this process and starts and stops the capture procedure. It takes the value 1 to start or 0 to stop the procedure, respectively.

| | |
|---|---|
| **<*axis*>.In1** | Returns the state of the drive digital input 1 (ON or OFF). |
| **<*axis*>.In1Mode** | Specifies the functionality of the drive digital input 1. The value 16 (keyword capturing) selects the capture input. |
| **<*axis*>.In2** | Returns the state of the drive digital input 2 (ON or OFF). |
| **<*axis*>.In2Mode** | specifies the functionality of the drive digital input 2. The value 16 (keyword capturing) selects the capture input. |
| **<*axis*>.In3** | Returns the state of the drive digital input 3 (ON or OFF). |
| **<*axis*>.*In3Mode*** | Specifies the functionality of the drive digital input 3. The value 16 (keyword capturing) selects the capture input. |

## 4.2.2. Homing

Most machine-axes need to establish a soft home reference position. This user-specified homing program is generally run with the power-up cycle of the machine or with a manual-mode. The soft home reference position (0000….) is established by a homing program for the hard-home plus any marker-pulse (for the integrity between the controller and axis-mechanics) with **HOMEOFFSET** (the distance from the hard home).

The hard-home position on a direct drive axis is usually assigned by the once-per-revolution marker pulse of the encoder (incremental, sine encoder or ENDAT encoder) or the zero-crossing position of a resolver (see **HOMETYPES**). However, for the machine-axis with a gear-ratio (geared, belted, or ball screw, etc.), a more complicated procedure is required for the relationship (integrity) between the controller and axis-mechanics of the homing program. In these cases, it is common to place a micro-switch or proximity-switch on the axis for the indication of the hard-home-reference position.

**NOTE**

***This home-position switch is placed on one end of the axis (inside the mechanical end of the travel-limits of the axis) with the switch tripping mechanism designed such that switch-circuit is closed on one side of the hard home-position and open on the other side of the hard home-position.***

The switch-circuit (high/low) provides the controller-input information for the direction of the home position.) Other machine axes may use absolute-position-device designs, so the integrity between the controller and axis mechanics is established automatically on the power-up of the machine (**HOMETYPES**). The machine position-integrity is established similarly to an axis that only rotates 360°.

The drive executes the **HOME** procedure. When you start the homing procedure on an axis **,** the drive immediately starts executing the homing procedure. The axis properties used to control the homing procedure are:

| | |
|---|---|
| **<*axis*>.HomeAcceleration** | Specifies the acceleration and deceleration to be used during the homing procedure (with the inclusion of **HOMEOFFSET** or **HOMEDISTANCE**). |
| **<*axis*>.HomeDirection** | Specifies the direction of the home search movement. Application-dependant instruction for the model: **HOME** to a hard-stop requires a movement in one direction. However, home to a switch-transition at one end of the axis-travel requires the direction for a closed-switch to be opposite that of an open-switch. |
| **<*axis*>.HomeDistance** | Sets the distance from the soft-home (position: 0000…. within the MC, see **HOMEOFFSET**) and the start position for any given program as you define. `A1.HomeDistance=5000` commands the axis to go to position 5000 from the soft home (position: 0000….) and stay there. |

| | |
|---|---|
| <***axis***>**.HomeDistanceMax** | Specifies the maximum distance from the point at which the homing process began (in either direction) that an axis is allowed to travel in order to find the hard-home position-switch-transition. |
| <***axis***>**.HomeType** | Specifies the homing latch type: 0=Switch + marker1=switch2=marker |
| <***axis***>**.HomeOffset** | Sets the soft home position to position 0000…. within the MC after **HOMEOFFSET**-specified distance has been traveled from the hard-home position (micro or prox-switch, etc.). When maintenance on an axis effects the integrity between the controller programs and the mechanical axes (e.g., replacement of the homing-switch), it is the value of **HOMEOFFSET** that is adjusted for the re-establishment of the integrity. Actual programs are not effected once the **HOMEOFFSET** has been adjusted. See **HOMEDISTANCE** when there is more than one program that needs to repeatedly start from different positions. |
| <***axis***>**.HomePolarity** | Specifies the homing switch edge1=rising edge2=falling edge |
| <***axis***>**.HomeReturn** | Specifies whether the axis assigns and adjusts to the proper position feedback for the home-position as the axis travels through the home switch-transition or go home (back to the switch after the first switch-transition for the minimization of any position-lag error + any **HOMEOFFSET** or **HOMEDISTANCE** for the assignment of the home-position)0 = do not go back1 = go back to the switch |
| <***axis***>**.HomeStatus** | Provides the status of the home (read only property)0 = axis not home1 = axis is home2 = home in progress3 = error during homing procedure |
| <***axis***>**.HomeVelocity** | Specifies the velocity to be used during the homing procedure (with the inclusion of **HOMEOFFSET** or **HOMEDISTANCE**). |

The various homing properties are provided to allow you to set the values of the parameters that determine the behavior of the procedure. For instance, properties such as <***axis***>**.HOMEVELOCITLY**, <***axis***>**.HOMEACCELERATION**, <***axis***>**.HOMEDISTANCE**, and <***axis***>**.HOMEOFFSET** allow you to control the rate at which the axis approaches home.

The following example illustrates this procedure: A typical home application may look like:

```
a1.HomeVelocity = 100
a1.HomeAcceleration = 1000
a1.HomeOffset = 100
a1.HomeDistance = 100
a1.HomeDirection = 1
a1.HomeType = 1
a1.HomePolarity = 2
a1.In1Mode = homing
a1.HomeDistanceMax = -1
a1.HomeReturn = 1
Home a1
```

For more information about these properties, refer to the *SERVO**STAR**® MC Reference Manual.*

# 4.2.3. Registration

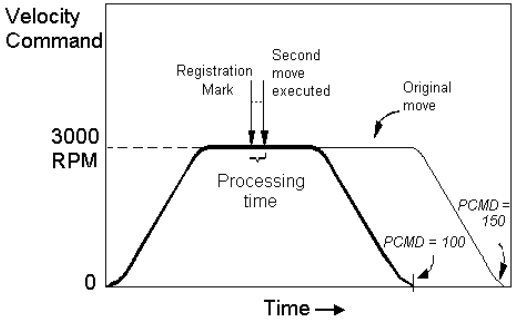
Registration applications are those that start motion and then modify the profile, based on a subsequent event. These applications generally involve discrete product processing, such as are common in the packaging, printing and converting industries. The event is usually generated by sensing a position on the product being processed (usually 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 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 MC combines the ability to change the position end points on-the-fly and the ability to command the SERVO**STAR** drive 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. The following example shows a typical registration example:

```
Program
'Prepare SERVOSTAR drive to capture
VCruise = 3000
Cycle:
Move FeedAxis 50
'wait for capture
Move FeedAxis [capture position] + 10 StartType=IMMED
End Program
```

This example generates the next profile.




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 comes to rest at a fixed offset after the registration mark.

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 ft/s, and the capture accuracy is ±3 microseconds, the mark is:

10 (ft/sec) * 3 µs = 3.225e-5 ft or about 0.0004 in.

The original line was set to go much farther than the actual resultant move. This technique is commonly used because you can monitor the end position and, if the move is completed before the registration mark is detected, it indicates the mark was missed.

## 4.2.4. Gating

Many applications require motion to be gated to an external switch. In these cases, you need to start a move after the gating input transitions with as little delay as possible. One way you can accomplish this on the MC is by starting the move with events containing synchronized motion. For example:

```
Program
Attach MainAxis
Move MainAxis 100 Absolute=FALSE StartType=SYNC
While System.Din.1=False;  'Wait for input to change
 Sleep 1
End While
SyncStart MainAxis         'Start the axis up by disable
End Program
```

This program moves MainAxis 100 units when DIN.1 transitions high. Motion on MainAxis starts as soon after the transition of DIN.1 as possible. **OnEvent** can also be used.

## 4.2.5. Clamping

Clamping applications typically process discrete parts but the end point of a move is unknown. Clamping is often applied when the length of the part is not accurately known or when the thickness of material is unknown. In clamping, the controller moves an axis at low torque until the end of the material is detected. Then the torque level is increased, either to measure the part more accurately or to hold the part for another axis to process.

The MC allows you to change torque limits on the drive and uses events to monitor position error to sense the end of the piece (below).

```
Program
Dim Shared PartFoundFlag As Long
OnEvent PartFound Abs(ClampingAxis.pe ) > 0.25
Stop ClampingAxis
[Set torque for heavy torque]
Sleep 30                        'allow 30 ms for settling
PartFoundFlag=TRUE
Return
End OnEvent
eventon PartFound
[set torque for light torque]
PartFoundFlag=FALSE
Move ClampingAxis Vcruise=10
While (PartFoundFlag=FALSE)
 Sleep 10
End While
REM Process part here
Return
End Program
```

This program works by starting a slow move with low torque. The move continues until position error is greater than 0.25 units. When the axis runs into a stop, position error accumulates because the profile continues. You must be careful to set the threshold of the event greater than the position error of normal operation. The threshold of the event must also be smaller than the drive's maximum following error to avoid drive error before the event is triggered. Disable **OnEvent** during the acceleration of the move to avoid nuisance PartFound events.

## 4.2.6. PIPEMODE

The MC controller can be used as a pure SERCOS interface card between your application and SERCOS drives (PIPEMODE feature). The host computer controlls all of what normally occurs in the motion module of the MC. It must send a realtime stream of data through the MC to the drives. In this case, the MC functions as a data pipe. The feature is enhanced by the option of flexibly switching between pipe mode and regular mode. In typical applications, the MC starts in regular mode where homing, jogging and initial position adjustments are made. After that, the MC can be switched to pipe mode and the host computer takes control. There can be both MC-controlled and pipe-controlled axes at same time in the system. Following error and velocity over-speed errors are not checked with piped axes, but by the host.

An external profiler path can be designed and fed to the MC point-by-point. The controller functions only as a pipe and is not responsible for the correctness of the applied profiler. No system limits are checked except for position error and feedback velocity limitation. While running under this mode, no other movements are allowed.

Command points are written through the standard Fast-Data DPRAM interface and transmitted into the controller. The DPRAM is scanned every SERCOS cycle and the command is executed.

There can be up to eight (8) axes in the system operating in pipe-mode. The Fast-Data interface can be configured in one of two communication protocols using **SYS.PIPEMODE**: 1 – sending position command only, 2 – sending both position and velocity commands. When the system's pipe mode uses only position commands, the controller computes the current velocity. Depending on the pipe mode configuration, the controller limits the number of axes that can be operated in this mode. When system uses only position command, eight (8) axes can operate. When using both position and velocity values, only four (4) axes can operate.

### *4.2.6.1. DATA STRUCTURE*

The host CPU passes position and velocity commands via a data structure through the Fast-Data DPRAM. The API maps the data to **SYS.HOSTDOUBLE**. The SERCOS operation mode of every drive connected to an axis working in Pipe Mode must be configured (Position/Velocity/Torque). The drive operation mode is selected through a status word written in **SYS.VIN**.

**NOTE**

***Only drive modes previously configured in the SERCOS setup can be selected.***

The data structure mapping depends on the **PIPEMODE** state:

| MC variables | Byte offset | SYSTEM.PIPEMODE | |
|---|---|---|---|
| | | Position (1) | Position and Velocity (2) |
| Sys.HostDouble[1] | 0…7 | A1.PCMD | A1.PCMD |
| Sys.HostDouble[2] | 8…15 | A2.PCMD | A1.VCMD |
| Sys.HostDouble[3] | 16…23 | A3.PCMD | A2.PCMD |
| Sys.HostDouble[4] | 24…31 | A4.PCMD | A2.VCMD |
| Sys.HostDouble[5] | 32…39 | A5.PCMD | A3.PCMD |
| Sys.HostDouble[6] | 40…47 | A6.PCMD | A3.VCMD |
| Sys.HostDouble[7] | 48…55 | A7.PCMD | A4.PCMD |
| Sys.HostDouble[8] | 56…63 | A8.PCMD | A4.VCMD |
| Sys.Vin | 96…99 | A1…A8 operation mode info | A1…A4 operation mode information |

## 4.2.6.2. DRIVE OPERATION MODES

The drive's operation mode of every axis is set in a 2-bit field, where there are two active modes: 0 - position and 1- velocity (set in a 24-bit field **SYS.VIN** variable). Mapping of the **SYS.VIN** operation mode status variable is described below:

| Bits offset | SYSTEM.PIPEMODE | |
|---|---|---|
| | Position (1) | Position and Velocity (2) |
| 0, 1 | Operation mode of drive1 | Operation mode of drive1 |
| 2, 3 | Operation mode of drive2 | Operation mode of drive2 |
| 4, 5 | Operation mode of drive3 | Operation mode of drive3 |
| 6, 7 | Operation mode of drive4 | Operation mode of drive4 |
| 8, 9 | Operation mode of drive5 | NA |
| 10, 11 | Operation mode of drive6 | NA |
| 12, 13 | Operation mode of drive7 | NA |
| 14, 15 | Operation mode of drive8 | NA |

Every sample, the controller sends the operation mode to the drive. Without any dependency at the drive's operation mode, the controller sends the position and velocity commands to the drive.

```
Program
   Sys.EN = Off
   Sys.PIPEMODE = 2
' Definition As Position & Velocity mode
   Sys.EN = On
   Sys.Vin.1 = 1
' Velocity Operation Mode
    attach A1
   A1.PIPEMODE =1
' Pipe mode Activation
   A1.en=1
   While Sys.Din.1 = 1
' External condition for Pipe Mode
      sleep 1000
   end while
   A1.PIPEMODE = 0
' Exit from the Pipe mode
   Sys.Vin.1 = 0
' Return to the Position Operation Mode
   detach A1
End Program
```

# 5. MASTER-SLAVE

Master-Slave motion links the position of one axis (slave) to the position of another (master). The master axis is any axis controlled by the MC or an external axis. External axes are connected to the MC via the external encoder input on any of the drives controlled by the MC.

There are two types of master-slave motion: gearing and camming. Gearing is used when the position of the slave must 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.

## *5. 1 MASTER SOURCES*

The master signal for master-slave motion is one of three sources:

1. A physical axis controlled by the MC.
2. A simulated axis inside the MC.
3. An encoder signal connected to the external encoder input of a drive attached to the MC.

The internal MC clock produces an incremental pulse every SERCOS cycle. To set the master source of an axis, use **MASTERSOURCE**. For example, to set the master input for axis A2 to the position feedback of axis A1 enter:

```
A2.MasterSource = A1.PositionFeedback
```

This works whether A1 is a physical or simulated axis.

You can also use **POSITIONCOMMAND** (**PCMD**) as a source for gearing or camming. This eliminates the lag caused by the position error in the master axis. The negative side is the slave is following the command, rather than the actual position.

Many times, the master source needs to be an external encoder signal. This may be an encoder from a line-driven motor or from a servo motor controlled by an external device, such as a PLC or even another MC. Sometimes the master source is not a motor. For example, you can use a pulse train as a master signal. In all these cases, the signal is connected to the external encoder input of one of the drives controlled by the MC. The SERVO**STAR** drives accept standard quadrature encoder signals as well as pulse trains. Refer to the installation manual for your SERVO**STAR** drive for more information about the external encoder input.

After the external signal is connected, it can be assigned as the master source of any axis using the variable, **POSITIONEXTERNAL** (**PEXT**). For example, to set axis A2 to follow the external encoder input of axis A1, type:

```
A2.MasterSource = A1.Pext
```

For this function to work, the axis must be configured in SERCOS to run on telegram 7, and **PEXT** must be selected as one of the data elements communicated back from the drive.

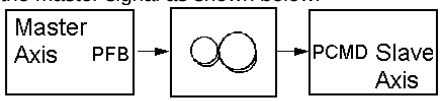
If the axis has to be controlled, instead of adding a simulated master, it can follow an internal clock. This makes the master a type of **TIMEPULSE**. Set the **GEARRATIO** to a low value (less then zero) and increase its value step by step to avoid exceeding velocity limit.

For optimal performance, the master axis should have an axis number less than the slave axis. If the slave axis is A1 and the master axis is A2, an additional time cycle is inserted in the motion.

If the master velocity override is changed, the slave velocity is changed accordingly. In a relative move, the slave moves according to its new velocity override. **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 moves added onto geared or cammed motion.

## *5. 2 GEARING*

When two axes are geared, the position command of the slave is proportional to the master signal as shown below.




MasterGearRatio

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

### 5.2.1. Enable Gearing

To enable gearing, set **MASTERSLAVE** to **GEAR**:

```
A1.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 and during gearing.

### 5.2.2. Disable Gearing

To disable gearing, set **SLAVE** to OFF:

```
A1.Slave = OFF             'Disable gearing
```

When gearing is disabled, the velocity of the slave axis is decelerated to zero at the rate of ***<axis>*.DECELERATIONMAX** (**DMAX**). Disabling an axis does not automatically disable gearing. You do not need to enable gearing again when the axis is re-enabled. However, issuing**JOG** or **STOP** for the axis disables gearing.

**STOP** turns gearing off immediately. The velocity of the axis is decelerated to zero at the rate set by ***<axis>*.DMAX**. For **JOG**, gearing is disabled immediately and the movement is controlled by the **JOG** profile.

If the master axis' RTK (**R**eal**T**ime **K**ernel or software code) is executed before the slaves' RTK, (i.e., if **?axislist** displays the master before displaying the slave) and the master source is **PCMD**, there is no delay in execution. If the master source is a feedback, **PFB** or **PEXT**, there is a system delay of 1 SERCOS cycle time. If the slave RTK is executed before the master RTK, an additional delay of 1 cycle time is incurred.

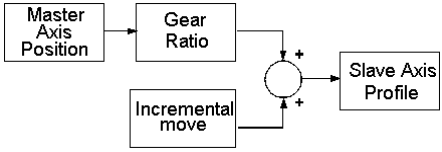
Set the proportional constant between the master and slave with ***<axis>*.GEARRATIO**:

```
A1.GearRatio = 0.5
'Slave goes half the speed of master
```

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

## 5.2.3. Incremental Moves

The MC supports incremental moves with gearing. The profile of the slave axis is the sum of two profiles: the gearing profile and the profile of the incremental move:



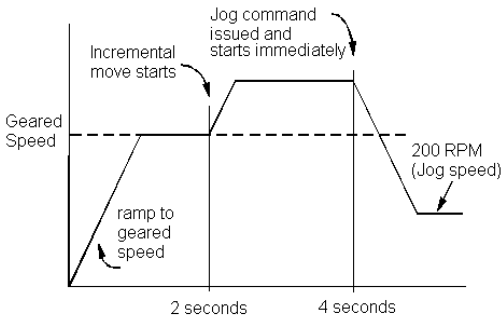

You can use incremental moves with and without gearing. Issuing an absolute move with gearing enabled generates an error. If you issue **STOP** to a geared axis executing incremental moves, gearing is turned off immediately and the velocity commanded from gearing ramps to zero at <***axis***>**.DECELERATIONMAX**. However, the termination of the incremental move profile and subsequent launch of **JOG** is subject to <***axis***>**.STOPTYPE**.

Issuing **JOG** when gearing with an incremental move is similar to issuing **JOG** with just gearing. The main difference is that the launch of the **JOG** profile is subject to <***axis***>**.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 either <***axis***>**.DEC** or <***axis***>**.ACC**. If **STARTTYPE** is GCOM, SYNC or INPOS, the jog profile is delayed.

The following example issues **JOG** with an immediate start (**STARTTYPE**=IMMEDIATE).

```
Jog A1 100 TimeJog=5000
Sleep 2000
A1.StartType=IMMED
Jog A1 200
```

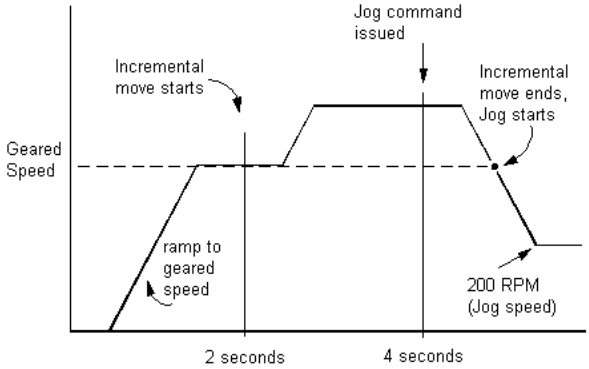
The resulting profile is:



This next example shows the previous example, but with the starting after the incremental move is complete (**STARTTYPE** = GCOM):

```
Jog A1 100 TimeJog=5000
Sleep 2000
A1.StartType=GCOM
Jog A1 200
```

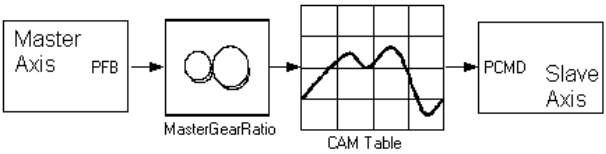
The resulting profile for this is:




## *5. 3 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 runs through a gear ratio and 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. The output of the cam table provides the slave axis position.

The operation of camming looks like:




### 5.3.1. Key Features

The MC's camming features are:

**Camming supports motion in both directions**. The master runs indefinitely in either positive or negative direction without accumulation of error.

**Cam tables are driven** by an axis position command, an axis position feedback, or an encoder brought in through the SERVO**STAR**'s external encoder input. It is driven by either physical or simulated axes.

**Cam tables are two-dimensional**. The points may be irregularly spaced. Linear interpolation is used between points.

**Cam profiles do not need to return** to the original position at the end of the profile.

**Cam tables are calculated** either off-line and stored, or on the fly.

**Cams cycle** either indefinitely or are specified to cycle a specific number of times.

**Multiple cams** are defined and linked together automatically.

**The master position is processed** through the master-slave gear ratio (**GEARRATIO**).

**Issued incremental moves are summed** with the cam profile to form the slave position command.

## 5.3.2. GEARRATIO

In camming, you set the proportional constant between the master and slave with <***axis***>**.GEARRATIO**. If GEARRATIO = 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:

```
A1.GearRatio = 0.5    'Drive cam table at half speed
```

The master turns 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. Its value is changeable during camming, but it has to be done carefully to avoid a position jump in the slave value due to large differences between the old and new value.

## 5.3.3. Incremental Moves

With incremental moves to the slave, the profile from the incremental move is summed with the profile for gearing to form the slave position.

## 5.3.4. Cam Tables

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

CAM1.1 = (10, 20)
CAM1.2 = (20, 40)
CAM1.3 = (30, 60)
CAM1.4 = (40, 80)
CAM1.5 = (50, 100)

In cam tables, the position of both the master and slave are in the position units of the respective axes. The individual points in the table are 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 is irregular, it must always be greater than zero. The master position must be monotonically increasing or decreasing in the cam table. The slave position is not limited by this requirement.

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

Cam tables are **incremental** in one aspect and **absolute** in another. Within the cam table, the positions are absolute. Suppose you want 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) is:

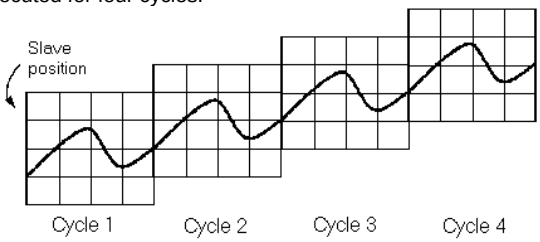
CAM1.1 = (10, 20)
CAM1.2 = (20, 40)
CAM1.3 = (30, 60)
CAM1.4 = (40, 80)
CAM1.5 = (50, 100)

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. Creating cam tables is covered later.

Cam tables are incremental with reference 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):

CAM1.1 = (0, 0)
CAM1.2 = (10, 20)
CAM1.3 = (20, 40)
CAM1.4 = (30, 60)
CAM1.5 = (40, 80)

Cam tables can be specified to command **net motion**. The start and end points of the cam table are 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 allowing it to be called many times in succession. Consider the following graph that shows just such a cam table with net motion executed for four cycles:




Each successive cycle ratchets the slave position up. The starting position in the table cannot be equal to the actual slave position or the cam slave position (which is fixed in the cam table) and the actual slave position (which is ratcheting up each cycle) does not match for more than one cycle.

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 **GEARRATIO**.

In many applications, you must ensure that this quantity is precise or the master appears to creep over time. The MC uses double-precision math (16 digits of accuracy) for cam calculations. Use the MC to calculate mathematical expressions to obtain the greatest possible precision. For example, if GEARRATIO=1:3, use the MC 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, you must assign the slave positions of the first and last point-pairs to be identical.

At the point when camming is enabled, the MC takes a snap-shot of the master position feedback, position or position external (depending on master definition), and slave position 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, the master position is 1000 and the slave's position (**PCMD**) is -2000. Each point-pair in the table is implicitly adjusted by (1000, -2000) throughout the cam cycle.

After each cam cycle, the offset to the master and slave is adjusted by the MC. 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.

Both the master and slave can be in **rotary mode** without affecting the camming operation.

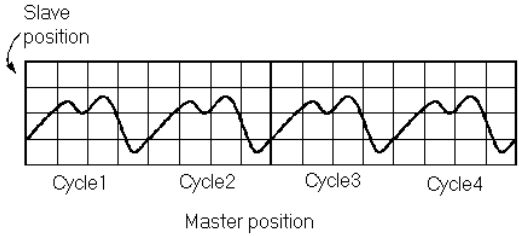
## 5.3.5. Cam Cycles

One cam cycle is the operation of moving the master position feedback through an entire cam table. Instead of using master position feedback, you can use position or posistion external. At the end of a cam cycle, the cam ends or transitions to the next cycle. The next cycle simply repeats the current cam or is another cam. The number of times a cam is repeated is controlled by <***cam***>**.CYCLES**. Other cams are linked in automatically with **NEXT** and **PREVIOUS**.

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. Then, the cam moves to the next cycle. Similarly, when the master is counting down, the cycle ends when master **PFB** is less than the first entry in the table. 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 and the slave axis stops.

Whether or not the cam cycle repeats, and the number of times it should repeat, is specified in <***cam***>**.CYCLES**. You can enter **1** to indicate that the cam runs once and does not repeat, **-1** to indicate the cam runs indefinitely, or use a positive numberto indicate the number of times the cycle runs.

For example, suppose you want a cam to run four cycles as shown below:




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, use:

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

When two cam tables are interconnected in a double-linked chain, only one cam table is active at a time. This enables changes in the other chained cam table by controlling the cycle time parameter of the active cam table. Setting its value to –1 results in an infinity periodic of the active cam. After updating all the relevant values in the chained table, set the active table cycle to 1. At the end of the current period, the two tables are exchanged and the chained cam table is generated.

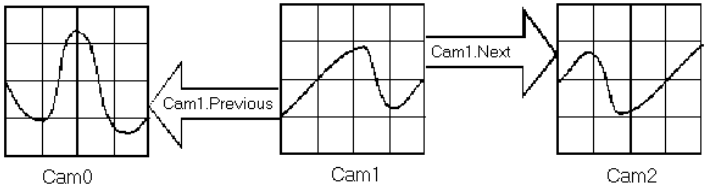
You can monitor the number of cycles run with **<*axis*>.CAMCYCLE**. When the master is moving forward, **CAMCYCLE** counts up. When **CAMCYCLE** reaches **CYCLE**, the axis transitions to the next cam, if there is one. Similarly, when the master is running backward and **CAMCYCLE** reaches 0, the cycle transitions to the previous cam, if there is one.

The MC allows you to specify multiple cam tables at the same time. The maximum number of cam tables is 256. The MC allows you to link cam tables together using two properties of the current cam table: **NEXT** and **PREVIOUS**. These properties allow 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 **NEXT** and **PREVIOUS** to the desired cam table. For example:

```
StartCam.Next = ForwardCam
StartCam.Previous = ReverseCam
```

The linking of the three cams (Cam0, Cam1, Cam2) is shown in the next figure.




Set **NEXT** and **PREVIOUS** independently. The **NEXT** and **PREVIOUS** pointers cannot be changed dynamically. The change takes 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, set **NEXT** and **PREVIOUS** to NULL (0).

If you have linked cams, the cam cycle runs the number of times specified in **CYCLES**. For example, assume Cam1 is being driven by a forward rotating master. If CAM1.NEXT=CAM2 and CAM1.CYCLES=4, Cam1 cycles four times before transitioning to Cam2. Similarly, if the master is rotating backward and CAM1.PREVIOUS=Cam0, Cam1 cycles four times before transitioning to Cam0.

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.

## 5.3.6. Create Cam Tables

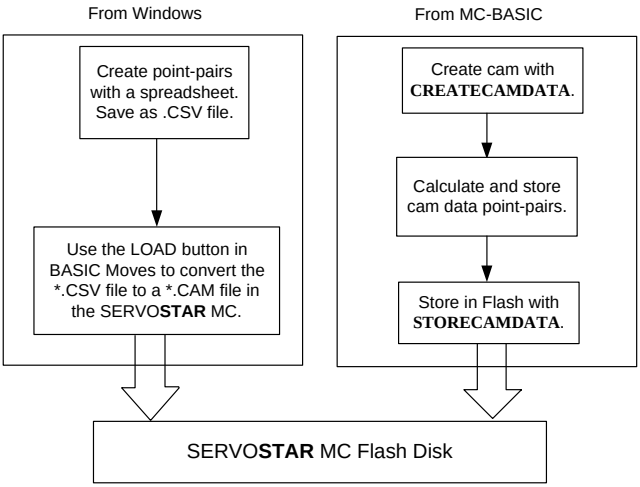
Cam tables contain the point-pairs that map the master position feedback to the slave position. Cam tables are only a part of cams. Other properties of cams include **CYCLE**, **NEXT** and **PREVIOUS**. Only the table portion of cams are stored permanently in the MC's Flash disk. The other properties are 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 while dynamic tables are built at run time. You should choose a type based on your application.

### *5.3.6.1. STATIC TABLES*

Static cam tables are built offline and stored permanently in the MC's Flash disk. They allow 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, then use MC tools to convert and store the table. Second, you can build static cam tables inside the MC. Both methods are shown in the next figure.



If you are using a static table, use the Windows-based tools. This method is simpler and you have more tools available with which to graph and analyze the cam table.

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 is the master position. The second is the corresponding slave position. 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 stay the same.

When the table is complete, save to a Comma Separated Variable or .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 looks like:

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

When you add the .CSV file to your project, BASIC Moves converts it to a cam table and stores it in the MC.

When building a cam table using MC-BASIC, first globally allocate cam storage. Use **Common Shared** <***var***> **as Cam**:

```
Common Shared CAM1 as Cam
```

Here, CAM1 is used as the name. Replace CAM1 with any legal MC name. The easiest way is to include this line in Config.Prg and reboot the MC. The alternative is to enter this at the terminal window. Remember that the effect of this line persists only until the MC resets.

Using MC-BASIC, create storage for the table of point-pairs. First, select a task to calculate the point pairs and enter the following line in that task:

```
Program CamMaker
CreateCamData 5 CAM1
```

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

Calculate the point-pairs to use in MC-BASIC. Build loops and iterate through the array. The array is 1-based and you 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
```

Finally, store the table on the flash disk. If you use:

```
StoreCamData MyCam.Cam Cam1
```

This stores the cam as MyCam.Cam in flash memory. The name of the file follows the 8.3 format. The filename cannot have more than 8 characters in the main part and 3 characters in the extension. The cam data file must have the .cam extension This file can be loaded at any time using **LOADCAMDATA**.

If you created the cam storage (**Common Shared <*var*> as Cam**), delete it with:

```
DeleteCam Cam1
```

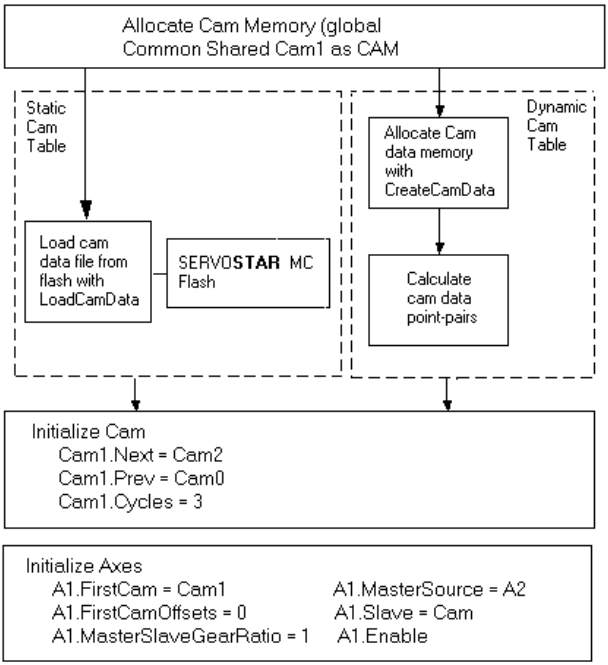
A cam table cannot be deleted if it is in use or if the task that it is attached to is loaded.

### 5.3.6.2. *DYNAMIC TABLES*

Cam tables can 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 place the data in Flash. Instead, you use the cam table directly.

## 5.3.7. Operating Cams

Now that you know the concepts of camming, you need to know the step-by-step process to use one. Throughout this section, CAM1 is our example. The entire process is shown in the figure below:



**Step 1: Allocate Space** Allocate space for the cam with **Common Shared** <***var***> **as Cam**. All cams are global, so this line must be located in Config.Prg. For example:

```
Common Shared CAM1 as Cam
```

**Step 2: Load Cam Data** This can be accomplished in two ways. For a static cam data array which has been calculated and stored in the MC, use **LOADCAMDATA**:

```
LoadCamData MyCam.Cam CAM1
```

or dynamically calculate the cam. This is a two step process. First, create space for the data array and then fill that space. For example, for a five element array, you 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. The difference is that this cam table is never stored to or loaded from flash.

**Step 3: Initialize Other Elements** Initialize other elements of the cam:

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

**Step 4: Initialize the Axis** Initialize the elements of the axis related to camming. Set **FIRSTCAM** to the first cam the axis follows:

```
A1.FirstCam = CAM1
```

Set the master position within the cam at the point where you wish to start. For example, you may have a cam table where the master position goes from 0 to 100.00, but the correct starting position for the cam might be 33.333, so you enter:

```
A1.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 is generated if you do not set **FIRSTCAMOFFSET**.

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

```
A1.GearRatio = 1.00
A1.MasterSource = A2.PFB
```

Finally, set **SLAVE** to CAM:

```
A1.Slave = CAM
```

**Step 5: Enable the Axis** At this point, camming is enabled. However, you must enable the drive to see motion:

```
A1.Enable = ON
```

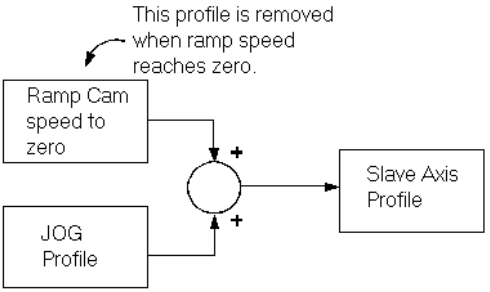
Camming cannot be enabled if the axis is in a relative or absolute move, or during **STOP**. Enabling camming when gearing is enabled or when an absolute move is being executed generates an error. Disabling or enabling the drive does not affect whether gearing is enabled.

To disable camming, set **SLAVE** to OFF:

```
A1.Slave = OFF                    'Disable camming
```

When camming is disabled, the velocity of the slave axis is decelerated to zero at the rate of <***axis***>**.DECELERATIONMAX** (<***axis***>**.DMAX**). Issuing **JOG** or **STOP** for the axis disables camming. Issuing **STOP** turns camming off immediately. The velocity of the axis is decelerated to zero at the rate set by <***axis***>**.DECELERATIONMAX**.

If **JOG** command is issued, camming on that axis is disabled immediately and the cam profile ramps to zero at the rate, <***axis***>**.DEC**. The **JOG** profile ramps up at the rate specified by <***axis***>**.ACC**. For a short time, the axis profile is the sum of these two profiles as shown below.




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 is commanded accordingly, when re-enabled. A fault is generated when the drive is re-enabled if the distance between the slave position and the command from the cam table is greater than **POSITIONERRORMAX**. Be cautious when re-enabling a cammed slave axis if a large position error exists. The motor rapidly upon enable to immediately correct this error.

**Step 6: Read Dynamic Information** There are numerous read-only properties regarding the operation of the cam.

**Axis Properties**

***<axis>*.ACTIVECAM** provides the name of the cam being executed by that axis.

**Cam Properties**

***<cam>*.CAMCYCLE** 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 CAMCYCLE=CYCLE.

When the cam master is moving backward, the current cam ends at the completion of the cycle where CAMCYCLE=0.

***<cam>*.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=50.

***<cam>*.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]=0.00.

CAM1.MASTERDATA[CAMINDEX] is not equivalent to the position of the master signal (<***axis***>**.PFB**) because cams are incremental with respect to starting position.

***<cam>*.SLAVEDATA** 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 <***axis***>**.PCMD** because the two may be offset from each other.

# 5. 4 SIMULATED AND MASTER-SLAVE AXES

Simulated axes are used to enhance machine control. They typically are with gearing and camming. Three common uses are:

1. To act as fixed speed masters for slaved axes.
2. To monitor physical axes.
3. To synchronize slaved axes.

## 5.4.1. Fixed-Speed Masters

The most common use for a simulated axis is as a fixed-speed master axis for geared or cammed axes. This is used for testing or as part of normal operation. The simulated axis is normally set to a fixed speed using **JOG**. Slave axes are driven without a physical motor. For example, if you want to view a cam table, you can use a simulated axis running at a fixed speed. In a sense, the simulated axis provides the ideal condition where the master speed is precisely fixed. This can allow you to fine-tune cams and carefully inspect the synchronization of geared axes.

Fixed-speed simulated axes and cams are combined to allow you to produce virtually any profile you need for a machine. You need only load the profile, cam table and drive the axis with a fixed-speed simulated axis.

## 5.4.2. Monitor Physical Axes

Simulated axes are used to monitor physical axes for machine control by slaving a simulated axis to a physical axis and monitoring the position and/or velocity of the simulated axis. For example, if you slave a non-rotary simulated axis to a rotary physical axis, you can monitor the total distance the physical axis has traveled because any variable from a simulated axis can be used to fire events. You can generate events based on the simulated axis. The position feedback of a simulated axis can drive a Programmable Limit Switch (PLS) to add more monitoring flexibility.

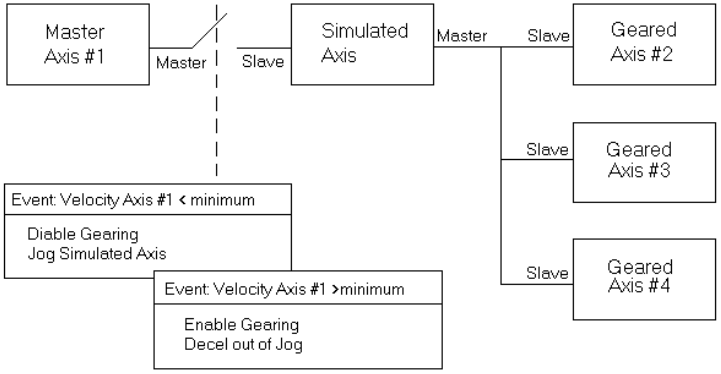
## 5.4.3. Synchronize Slave Axes

Many machines have multiple slaved axes that are controlled as a group. Normally, synchronization is maintained when these axes share a single physical axis as a master. However, on some machines, there are special modes of operation where a physical axis cannot provide all the necessary functions. Consider a machine where a minimum speed must be maintained on the slave axes even if the master axis stops. In this case, you can insert a simulated axis between the physical master and slave drives. To do this, slave a simulated axis to the physical master and then slave the other axes to the simulated axis. Use events to tell when the physical axis is above or below the minimum and use **JOG** to keep the simulated axis above the desired minimum to keep the slave axes moving even if the physical master stops.

For example:

```
Program
   OnEvent Vel_Below_Min PseudoAxis.Vcmd < 100
   Jog PseudoAxis 100
'This indirectly turns gearing off
   End OnEvent
   OnEvent Vel_Above_Min PseudoAxis.Vcmd >= 100
   PseudoAxis.Slave=Gear
   End OnEvent
   Attach PseudoAxis
   EventOn Vel_Below_Min
   EventOn Vel_Above_Min
   While 1
       Sleep 100
   End While
End Program
```

The following figure shows this process.



# 6. GROUPS

Groups allow you to control multiple axes as a single mechanism. With groups, the position command and feedback signals are no longer single values, but instead, are vectors with two or three elements. Velocity and acceleration no longer apply to a motor, but instead, apply to the combination of two or three motors moving in concert.

## 6. 1 SET UP

Groups are combinations of axes. These axes must be orthogonal if their scalar properties (velocity) are to be correct. For example, a group can be used to drive an x-y table. Groups are supported for two- and three-axis machines. In fact, any number of axes can be added to a group.

To set up a group, first set up each of the axes that will be in the group. Then, use **Common Shared** to form the group:

```
Common Shared MyGroup as Group AxisName = xAxis AxisName = yAxis
```

There is no limitation to the number of axes in a group. All axes in the system (including simulated axes) may be part of one group.

It is necessary to set **VELOCITYFACTOR**, **ACCELERATIONFACTOR**, and **JERKFACTOR** for the group. As a first approximation, set <*group*>.VFAC = 1/1000, set AFAC=VFAC/1000, and set JFAC=AFAC/1000.

## 6. 2 DELETEGROUP

**DELETEGROUP** deletes the specified group. The participating axes thereafter function only as independant axes. They must not be moving. No tasks may be loaded.

```
DeleteGroup MyGroup
```

## 6. 3 ACCELERATION PROFILES

As with single axes, the MC provides controlled acceleration profiles for groups. This reduces vibration and wear on the machine, and allows the highest acceleration rates. You can independently control the acceleration and deceleration rates.

### 6.3.1. Acceleration and Deceleration

<***group***>**.ACCELERATION** and <***group***>**.DECELERATION** control the acceleration rates of the group mechanism. For example, the following lines change the acceleration rates for MyGroup:

```
MyGroup.Acceleration = 1000
MyGroup.Deceleration = 1000
```

<***group***>**.ACCELERATION** and <***group***>**.DECELERATION** apply to the combined motion of the individual axes. For example, if MyGroup is a three-axis group formed with Axis1, Axis2, and Axis3, then:

```
MyGroup.Acc = (Axis1.Accel2 + Axis2.Accel2 + Axis3.Accel2) ½
```

Group acceleration is the orthogonal combination of the axes' acceleration. Group acceleration and deceleration are limited according to the axes' values.You can impose limits to constrain the time during which group acceleration and deceleration occurs. <***group***>**.TIMEACCELERATION** and <***group***>**.TIMEDECELERATION** define the duration of the acceleration phase.

## *6. 4 POSITION*

Group position differs from axis position in that a single property must represent the position of multiple axes. In some cases, the individual axis properties are maintained so that the property becomes a vector with two or three entries. This is the case for **POSITIONFEEDBACK** or **POSITIONCOMMAND**. In other cases, the position from multiple axes are combined using the square root of the sum of the squares, as for **POSITIONERRORMAX**.

### 6.4.1. Vector

With groups, three position properties are vectors: the position command, position feedback, and position final. In each case, the quantities in the vector are equivalent to the individual axis properties. For example, you can set **POSITIONFINA**L as part of a **MOVE**:

```
Move MyGroup {5.5, 6.5}
```

This sets **PFINAL** of the first axis to 5.5 and the second to 6.5. Querying the position feedback from a two-axis group from the terminal also returns a vector. For example:

```
? MyGroup.PFb
```

returns:

```
(5.600443e+003, 6.422422e+003)
```

The positions in these vectors are ordered according to the way the group was created with **Common…As Group**. The first axis declared occupies the first position in the vector, the second occupies the second, etc.

### 6.4.2. Scalar

Other group position properties are scalar. These properties are the orthogonal combination of the axis properties. These properties include:

**POSITIONERROR**
**POSITIONERRORMAX**
**POSITIONERRORSETTLE**
**POSITIONTOGO**

If MyGroup is formed with Axis1 and Axis2, and AXIS1.POSITIONTOGO = 1 and AXIS2.POSITIONTOGO=1, MYGROUP.POSITIONTOGO = 1.414, the orthogonal combination of the two axes' POSITIONTOGO. Similarly, when you set **POSITIONERRORMAX** or **POSITIONERRORSETTLE**, the position error they are compared to is the orthogonal combination of the axes' **POSITIONERROR**.

## 6. 5 VELOCITY

A group's scalar velocity properties are always the orthogonal combination of the respective axes' velocity properties. Group velocity properties include:

| | |
|---|---|
| **VELOCITYCOMMAND** | vector property |
| **VELOCITYFEEDBACK** | vector property |
| **VELOCITYCRUISE** | scalar property |
| **VELOCITYFINAL** | scalar property |
| **VELOCITYMAX** | scalar property |
| **VELOCITYOVERRIDE** | scalar property |

Each of these properties operates as its axis equivalent and are clearly defined in the *SERVO**STAR**® MC Reference Manual*.

## 6. 6 LIMITS

Group limits are similar to limits placed on axes. Limits can be imposed in two ways: they can be checked realtime, or they can be checked only for subsequent actions.

### 6.6.1. Generator

Generator limits affect subsequent commands to the motion generator. For example, if you change an acceleration limit, this affects subsequent motion commands but has no effect on the current motion.

### 6.6.2. Realtime

The MC checks realtime limits each SERCOS update cycle. For example, exiting position error of every axis is monitored each SERCOS cycle.

### 6.6.3. Position

Position limits are imposed solely by the axes and are checked at the beginning of every movement. If one of the target's points exceeds its limit, the whole movement is rejected. There are no group position limits. There are also several limits in the MC related to position:

Set **POSITIONERRORMAX** (**PEMAX**) to the total position error the system can tolerate during operation (maximum square root of the sum of squares):

```
MyGroup.PEMax = 1.0
```

Set **POSITIONERRORSETTLE** to the total position error the system can tolerate to be considered in position.

```
MyGroup.PositionErrorSettle = 1.0
```

**TIMESETTLE** (**TSETTLE**) is the amount of time required before an axis is considered settled (POSITIONERROR<POSITIONERRORSETTLE). After a **MOVE/CIRCLE** profile is complete, the MC waits for POSITIONERROR < POSITIONERRORSETTLE for **TIMESETTLE** milliseconds before setting **ISSETTLED**.

**TIMESETTLEMAX** (**TSETTLEMAX**) defines the amount of time allowed for settling after a move has been commanded. After the **MOVE/CIRCLE** profile is complete, the MC waits **TIMESETTLEMAX** milliseconds for settling. If the position error remains above **PESETTLE**, an error is generated.

**ISSETTLED** is a binary (ON or OFF) property that indicates if the group is settled. To be settled, the motion profile must be complete and the orthogonal combination of axis position errors must be below **POSITIONERRORSETTLE**. The time condition must also be satisfied.

### 6.6.4. Velocity

There is one group limit in the MC related to velocity: **VELOCITYMAX** (**VMAX**) sets the maximum speed the motion generator can command the group. This property is checked at the beginning of each move. For example:

```
Group.VelocityMax = 5000
```

The actual executing velocity is also limited according to the group's axes velocity limitations. The maximum allowed axes velocity is initialize group velocity. In this case a note is returned and can be seen only if **SYS.MOTIONASSISTANCE** is on.

## *6. 7 ACCELERATION*

Three limits control acceleration, deceleration, and jerk, which limit the velocity transients on acceleration profiles.

**ACCELERATIONMAX** is the upper limit for acceleration in MC acceleration units.

**DECELERATIONMAX** is the upper limit for deceleration in MC acceleration units.

The actual executing acceleration and deceleration are limited also according to the group's axes limitations. The maximum allowed axes of those parameters initialize the group's values. In this case, a note is returned and can be seen only if **SYS.MOTION ASSISTANCE** is on.

The jerk properties, **JERK**, **JERKFACTOR**, **JERKMAX**, and **SMOOTHFACTOR** are applicable in group motion to obtain smooth motion control.

## *6. 8 VELOCITY, ACCELERATION, DECELERATION AND JERK RATES*

**GROUP.VELOCITYRATE** defines the group velocity maximum scaling factor from 0.1 to 100 independent of acceleration, deceleration or jerk. **GROUP.VELOCITYRATE** may be modal or nodal.

**GROUP.ACCELERATIONRATE** defines the group acceleration maximum scaling factor from 0.1 to 100 independent of velocity, deceleration or jerk. **GROUP.ACCELERATIONRATE** may be modal or nodal.

**GROUP.DECELERATIONRATE** defines the group deceleration maximum scaling factor from 0.1 to 100 independent of velocity, acceleration or jerk. **GROUP.DECELERATIONRATE** may be modal or nodal.

**GROUP.JERKRATE** defines the group maximum Jerk scaling factor from 0.1 to 100 independent of velocity, acceleration or deceleration. **GROUP.JERKRATE** may be modal or nodal.

For example:

```
Common Shared Gr1 as Group AxisName=A1 AxisName=A2
Gr1.VRate = 50                    ‘ VelocityRate  50%
Gr1.ARate = 70                    ‘ AccelerationRate 70%
Gr1.DRate = 80                    ‘ AccelerationRate 80%
Gr1.JRate = 60                    ‘ JerkRate 60%
```

## *6. 9 MOTION*

A group is controlled by the motion generator. This software device receives commands from MC tasks and produces position and velocity commands for each drive in the group every SERVO cycle. The two group motion commands are **MOVE** and **CIRCLE**. The MC also provides synchronization of the execution of motion commands as well as changing motion profiles on-the-fly and blending a followed movement. In addition, the MC provides multiple modes for starting and stopping motion. Several conditions must be met before the MC generates group motion:

1. The controlling task must **ATTACH** to the controlled group.
2. All axes in the group must be **ENABLE**d.
3. The system and axis motion flags must be **ENABLE**d.

You can declare a group from every context. However, using Config.prg in most applications requires this to only be done once.

```
A1.Name = xAxis
A2.Name = yAxis
Common Shared xyTable as Group AxisName = xAxis AxisName = yAxis
```

### 6.9.1. Attach Task and Axis

Before a task can send motion commands to a group, the group must be attached to the task. This prevents other tasks from attempting to control the group. To attach a group, issue an **ATTACH** from the controlling task:

```
Attach MyGroup
```

If no other task is attached to the group and no axis from the group is attached, the group is immediately attached. Otherwise, an error is generated.

As long as the task has the group attached, no other task can control the group or axes within the group. The axes belonging to the group cannot be controlled independently from other tasks. When a task is finished with a group, it should detach the group:

```
Detach MyGroup
```

If a task ends, all attached groups and axes are automatically detached. One exception for the requirement that a group be attached is if the motion command is issued from the terminal window. In this case, assuming the group is not attached by any other task, the group is automatically attached to the terminal window for the duration of the motion.

### 6.9.2. ENABLE

Now, enable the drives by typing the following in the terminal window:

```
System.Enable = ON
MyGroup.Enable = ON
```

## 6.9.3. Motion Flags

The last step in preparing a system for motion is to turn the motion flags ON. There are two motion flags: the system motion flag and the motion flags of each axis within the group. For example:

```
System.Motion = ON
A1.Motion = ON
A2.Motion = ON
A3.Motion = ON
```

There is a motion flag for the group that can be used to enable motion for all axes in the group. In many applications, a hardware switch should be tied to the motion flag. The MC does not have a hardware motion input, but you can use events to create this function. The following example demonstrates a task that uses external input SYSTEM.DIN.1 to control the system motion flag:

```
Program MotionSwitch
   OnEvent MOTION_ON System.DIN.1 = ON
      System.Motion = ON
   End OnEvent

   OnEvent MOTION_OFF System.DIN.1 = OFF
      System.Motion = OFF
   End OnEvent
End Program
```

## 6.9.4. STOP

**STOP** stops group motion in the motion buffer. In the command, you must specify the group:

```
Stop GroupA
```

Normally, **STOP** is set to stop motion immediately at the rate of **DECELERATIONMAX**. However, you can modify the effects of **STOP** with **STOPTYPE**. <***group***>**.STOPTYPE** can take four values:

**STOPTYPE = IMMED**
Stop group immediately at **DECELERATIONMAX** on each axis. The current path of the n\movement is not preserved.

**STOPTYPE = ENDMOTION**
Stop group at the end of the current motion.

**STOPTYPE = ONPATH**
Stop group immediately. Keep the stopping trajectory on the current motion path.

**STOPTYPE = ABORT**
Stop the current motion immediately without the ability to restore the stopped movements.

**STOPTYPE** defaults to IMMED. If STOPTYPE = ENDMOTION, the current move is completed before **STOP** is executed. If STOPTYPE = ONPATH, **STOP** is executed so the group stays on the path. Stopping is according to the current movement's deceleration and not at maximum deceleration to remain on the path.

## 6.9.5. PROCEED

After a **STOP**, you must **PROCEED** to clear the motion buffer. You can use **PROCEEDTYPE** to specify which way the motion generator should continue:

***PROCEEDTYPE = CONTINUE***
The motion generator continues the stopped command until complete.

***PROCEEDTYPE = NEXTMOTION***
The motion generator aborts the current move and goes directly to the stopped move in the motion buffer.

***PROCEEDTYPE = CLEARMOTION***
Clears the motion buffer. No motion executes. Default value.

***STOPTYPE = ABORT***
Current motion stops immediately but does not need a **PROCEED** to start the next motion. Only the current motion is stopped, the following commands are generated.

## 6.9.6. MOVE

**MOVE** for groups is almost identical to a **MOVE** for axes. There are two differences. First, **POSITIONCOMMAND**, **POSITIONFEEDBACK**, and **PFINAL** are group size vectors. Second, the profile, including **ACCELERATION** and **DECELERATION**, applies to all axes in the group. For example:

```
MyGroup.VCruise = 2000
Move MyGroup {100, 100}
```

moves to position (100, 100) with a cruise velocity of 2000.

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

**ABSOLUTE**
**STARTTYPE**
**ACCELERATION**
**ACCELERATIONRATE**
**DECELERATION**
**DECELERATIONRATE**
**SMOOTHFACTOR**
**VELOCITYCRUISE**
**VELOCITYRATE**
**JERK**
**JERKRATE**

**PFINAL** (end of a move), is always specified in **MOVE**. However, the meaning of **PFINAL** depends on <***group***>**.ABSOLUTE**. This allows point-to-point moves to be specified two ways:

***ABSOLUTE = TRUE***
Final position is specified as actual group 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 is the sum of **PFINAL** and **PCMD** at the start of the move. So:

```
xyTableGroup.Absolute =  TRUE
Move xyTableGroup {100, 200}
```

moves xyTableGroup to position (100, 200).

On the other hand:

```
xyTableGroup.Absolute =  FALSE
Move xyTableGroup {100, 200}
```

moves xyTableGroup a distance of (100, 200) units from the start position.

**ABSOLUTE** defaults to INCREMENTAL. You can change **ABSOLUTE** any time, although the effect does not take place until you issue the next **MOVE**.

## 6.9.7. CIRCLE

The MC supports 2-axis circular interpolation with **CIRCLE**. In **CIRCLE**, you specify:

Group name

Angle to rotate (in degrees, positive = counter-clockwise) along with Center of the circle (**CIRCLECENTER** or **CTR**).

Target point (**TARGETPOINT**) along with another point on the arc (**CIRCLEPOINT**).

```
Circle xyTable Angle=90 CircleCenter = {0,0}
```

or

```
Circle xyTable CirclePoint = {10,20} TargetPoint = {100,200}
```

A 2-axis circle is generated in a specified plane with incremental movement in the third axis. That generate a 3 axis circle movement but not helical.

```
Circle xyTable CirclePoint = {10,20,10} TargetPoint =
{100,200,20} circleplane= xy
```

In the above example, the circle is implemented in the XY plane along with the incremental change of the Z axis. Optionally, you can override the following group properties with **CIRCLE**:

**ABSOLUTE**
**ACCELERATION**
**ACCELERATIONRATE**
**DECELERATION**
**DECELERATIONRATE**
**JERK**
**JERKRATE**
**SMOOTHFACTOR**
**STARTTYPE**
**VELOCITYCRUISE**
**VELOCITYMAX**
**VELOCITYRATE**

For example, you can enter the following absolute and incremental circular moves:

```
Circle xyTable Angle=90 CTR = {0,0} Absolute = False
```

The value of **ANGLE** is not limited to 360. Greater values are specified to turn the motor more than once. The sign of the angle determines the direction of the motion. The value of **ANGLE** is not permanent. It exists only in the context of the command in which it is specified. Consequently, you can not query the value of **ANGLE**.

The value of **CIRCLECENTER** is not permanent. It exists only in the context of the command in which it is specified. Consequently, you cannot query the value of **CIRCLECENTER**.

**CIRCLETARGET** is a group size array. It specifies the target point of the circle. This array is not queried. **CIRCLE,** defined using **CIRCLETARGET,** is limited to 360 degrees.

**CIRCLEPOINT** is a position array that specifies a point on the arc. The arc is calculated using the **CIRCLETARGET** and these points. This array is not queried.

In case of 3 dimensions circle the active **CIRCLEPLANE** defines which group axes participate in the circle movement. The third axis is moved linearly. There are 3 plans that can be chosen: XY, XZ and YZ.

## 6.9.8. Chain Commands

You can chain group **MOVE**s and **CIRCLE**s using <***group***>**.STARTTYPE**. The following program makes a series of moves.

```
A1.Name = xAxis
A2.Name = yAxis
Common Shared xyTable as Group AxisName = xAxis AxisName = yAxis
```

From another task named MoveCircProgram:

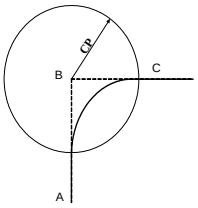
```
Attach xyTable
xyTable.Absolute = TRUE             'Move to start position
xyTable.StartType = INPOS           'Move to start and wait to settle
Move xyTable {-3, -2}

While(xyTable.isSettled = False)    'wait for settling
End While
'Begin Pattern
xyTable.StartType = GCOM    'Move smoothly from one move to the next
xyTable.Absolute = FALSE    'Incremental moves
Move xyTable {0, 4}
Circle xyTable Angle = -90 CircleCenter = {-2, 2}
Move xyTable {4, 0}
Circle xyTable Angle = -90 CircleCenter = { 2, 2}
Move xyTable {0,-4}
Circle xyTable Angle = -90 CircleCenter = { 2,-2}
Move xyTable {-4,0}
Circle xyTable Angle = -90 CircleCenter = {-2,-2}
While(xyTable.isSettled = False)      'wait for settling
Detach xyTable
End Program
```

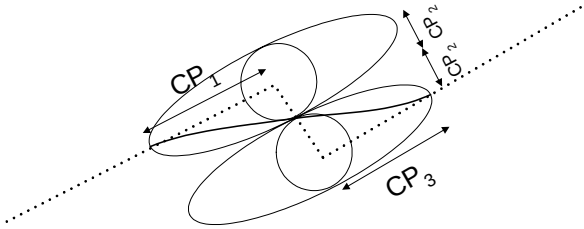
## 6.9.9. Blending

Two movements blend tighter to smooth connection points. There are two ways to blend the movements: preserving the movement's path (**CP**) or superposition. Set the blending method's ID with **BLENDINGMETHOD**.

The continuous path method (**CP**) blends where continuity and velocity smoothness are required (gluing, dispensing, …). It is less useful for high-speed pick and place applications where cycle time is the main quality measurement. Using this blending method, you ensure that maximum space velocities are never exceeded and the original path is always followed. The segment's middle point is always part of the blended path. Only two consecutive motions can be blended together.



When the second motion is completed before the first one, the system does not blend the first and the third motions together. The main parameter in this type of blending **CP**. It specifies the distance from the target point on which the blending (second motion) begins. **CP** is dimensional. It is expressed in group position units. The system automatically limits this value to half the segments length.




When the second movement is received too late (during the deceleration phase) or when the current motion causes excessive values of acceleration and velocity, blending between two movements is ignored.

For Move-to-Move blending, use:

```
Pos = start + I1 direction1 + I2 direction2
```

For Move-and-Circle blending, use:

```
Pos = start + I1 direction1 + (cos(I2)-1) rot_start2 + sin(I2)
rot_normal2
```

For Circle-and-Circle blending, use:

```
Pos = center1 + (cos(I1) rot_start1 + sin(I1) rot_normal1 +
(cos(I2)-1) rot_start2 + sin(I2) rot_normal2
```

## 6.9.9.1. SUPERPOSITION

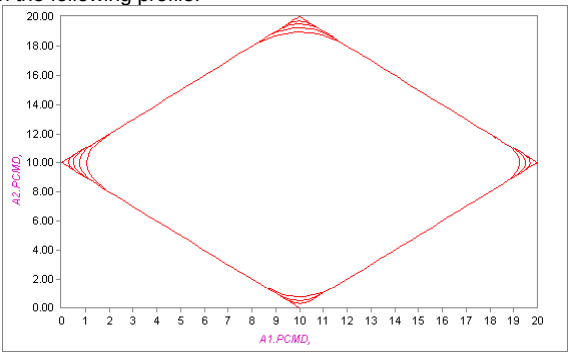
Superposition blends motions at any point of time starting from the initial point of the movement up to the target. This blending mode is not limited by the combined actions of the two motions. It is your responsibility not to blend together motions that, in combination, produce velocities or accelerations outside of allowable system limits. However, each individual motion is limited by the usual kinematics constrains (**VMAX**, **AMAX**, **DMAX**, **JMAX**). This blending method is also able to swap movements when the second is finished before the first. In this case, there is continuous blending of the first and the third movements. Although this is a feature, it must be taken into consideration when designing blending paths.

Using **BLENDINGFACTOR**, you define the beginning blending (second motion) point anywhere on the first movement path. It is dimensionless cy using a percentage of the path where the blending starts. A zero value of **BLENDINGFACTOR** designatess full blending, while 100 designates no blending.

For example.

```
Dim Shared blen_value As Long
Program
Attach xyTable
  BlendingMethod = 2
  BlendingFactor = 0
  StartType = GCom
  For blen_value = 100 To 80 Step -5
   Move xyTable {0,10} Abs=1 Vcruise=100 BlendingFactor=blen_value
   Move xyTable {10,20} Abs=1 Vcruise=100 BlendingFactor=blen_value
   Move xyTable {20,10} Abs=1 Vcruise=100 BlendingFactor=blen_value
   Move xyTable {10,0}  Abs=1 Vcruise=100 BlendingFactor=blen_value
  Next
  Detach xyTable
End Program
```

results in the following profile:




The following are blending limitations:

Blending type of the first movement defines the actual connection path, although the second movement has a different definition. The second movement can blend with the current movement, although its blending method was not defined. In this case, blending occurs according to the second movement.

Throughout the whole blending process, sine-acceleration or velocity trapeze profiles are used. However, mixing profile types is not allowed.

In the next example, no blending is ignored:

```
BlendingMethod = 0  CP = x BlendingFactor = x
BlendingMethod = 1  CP = 0 BlendingFactor = x
BlendingMethod = 2  CP = x BlendingFactor = 100
```

The Position Command Generator (PCG) has two working modes: single and double. When PCG is in single mode, only one profiler call per sample is allowed. This is identical to no blending. When PCG is in double mode, there are two profiler calls per sample that actually execute the motion superposition. If the profiler of the second motion is in Profiler Following mode, the profiler of the first movement leads both movements. Use **DOUBLEMODE** to detect double mode interpolation during blending.

## 6. 10 MOVE CONTROL

This section reviews options the MC provides for controlling group **MOVE**s and **CIRCLE**s.

## 6.10.1. Settling Time

The MC actively watches to determine if the groups are settled into position. In most applications, the group 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*.

Ideally, settling time allows the group to move so that position error is zero . However, in real systems you must allow for the condition where the position error never quite reaches zero. On the MC, you specify how low you consider to be low enough with **POSITIONERRORSETTLE** or **PESETTLE**:

```
xyTableGroup.PESettle = 0.01
```

After the motion generator completes a move that ends with zero speed, it actively monitors the position error of the group (the orthogonal combination of the position error of each axis) to determine when it is between **±PESETTLE**. If it is within this range, <***group***>**.ISSETTLED** is true (1). Otherwise, it is false.

In some applications, you must ensure that the position error remains below **PESETTLE** for a specified period of time before the group is considered settled. The MC allows you to specify this time period with the <***group***>**.TIMESETTLE** or <***group***>**.TSETTLE** (given in milliseconds). If the position error exceeds **PESETTLE** during this time, the timer is reset. If position error is within **PESETTLE**'s range for the time specified by **TSETTLE**, <***group***>**.ISSETTLED** is true (1). Otherwise, it is false. For example:

```
PESettle = 0.01
TimeSettle = 0.01 xyTableGroup.StartType = INPOSMove
xyTableGroup {100 , 200}
Move xyTableGroup {200 , 400}
```

This example requires that after the motion generator completes the first move and xyTableGroup has position error at or below 0.01 for at least 10 ms before the group is settled. Since **STARTTYPE** is INPOS, the motion generator waits for these conditions before starting the second move.

## 6.10.2. Start Moves

When the motion buffer is empty, group moves begin immediately. If you need to delay the next motion, you have two options. You can either use **DELAY** to insert a fixed delay time or **STARTTYPE** to delay, depending on a condition.

Use **DELAY** to force the motion generator to wait a fixed period of time between moves. The motion element xyTableGroup must be specified in **DELAY**. For example:

```
Move xyTableGroup {100, -100}
Delay xyTableGroup 1000
Move xyTableGroup {200, 500}
```

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

**DELAY** differs from **SLEEP** in that **SLEEP** delays program execution and **DELAY** delays only the motion generator. **DELAY** is forced to the **STARTTYPE** mode.

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

***StartType = GeneratorCompleted* or *GCOM***
When STARTTYPE=GCOM, the new move starts as soon as the motion generator completes the current move's motion.

***StartType = InPosition* or *INPOS***
When STARTTYPE=INPOS, the motion generator delays executing the new motion until the current move completes and the position error settles to near zero.

***StartType = Immediate (IMMED) or SuperImmediate (SIMM)***
When STARTTYPE=IMMED or SIMM, 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.

***StartType = Sync***
When STARTTYPE = SYNC, the motion is synchronized with **SYNCSTART**.

## 6.10.3. Chain Moves

When chaining multiple moves which all end at zero speed, you normally want STARTTYPE=INPOS. This forces the motion generator to wait for the position error to become small before starting the next move. If the position profile starts on the second move too soon, the motor may never come to rest. Normally, the desired performance is for the velocity to go to zero before the second move starts. This is the case where **STARTTYPE** = INPOS.

The ending speed of a **MOVE** defaults to zero. However, you can specify a speed other than zero as the end speed. To do this, <***axis***>**.VELOCITYFINAL**. For example:

```
MyGroup.VelocityCruise = 2000
Move MyGroup {1.5, -2.33} VelocityFinal = 1000
Move MyGroup {2.0, 4.2}
```

When the final velocity is not specified, the movement terminates with zero velocity. If you specify a non-zero final velocity, the smoothness of the motion is your responsibility.

If there are no movement commands to chain, the system stops motion with automatic braking and displays a message.

## 6.10.4. Multi-Step Moves

Combining non-zero end-point moves in the motion buffer produces multi-step moves. For example:

```
REM First Step
MyGroup.VelocityCruise = 2000
Move MyGroup {100, 400} VelocityFinal = 1000    'Second Step
MyGroup.StartType = GCOM
MyGroup.VelocityCruise = 1000
 Move MyGroup {200, 400}
```

MyGroup.StartType is set to GCOM. The second move begins after the first move is complete. You must use STARTTYPE=GCOM when chaining a move after non-zero velocity end- move. You cannot use STARTTYPE=IMMED or SIMM because the new move overrides the old move. When the movement ends with non-zero final velocity, the next movement is started at the endpoint of the first movement or immediately, if **STARTTYPE** of the next movement is IMMED or SIMM.

You can combine non-zero velocity end moves to produce profiles with as many steps as needed. However, there are a few restrictions:

**STARTTYPE** for all non-zero velocity end moves (that is, VELOCITYFINAL<>0) must be GCOM.

**VELOCITYCRUISE** and **VELOCITYFINAL** must be positive.

**PFINAL** of the succeeding move must be far enough in front of **PFINAL** of the current move so the profile is possible with the acceleration limits. An error is generated if this rule is violated.

The end position or the end velocity of a move in progress can be changed by setting STARTTYPE=IMMED and issuing the second move. This move command cancels the current move. For example issuing a move without changing the velocity:

```
MyGroup.StartType = IMMED
Move MyGroup {150, -400}
Sleep 3000
Move MyGroup {150, -800}
```

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

```
MyGroup.StartType = IMMED
Move MyGroup {150, -400} abs=1
Sleep 3000
Move MyGroup {150, -400} abs=1 VCruise = 1500
```

You can change the final position, cruise velocity, and final velocity of any executing move. Observe a few rules:

**STARTTYPE** must be IMMED or SIMM.

New and old **VELOCITYCRUISE** and **VELOCITYFINAL** must be positive.

If the succeeding move changes **PFINAL** or **VFINAL**, it must remain possible to create the profile with the axis acceleration limits.

## 6.10.5. Synchronize Multiple Axes

The MC provides the ability to synchronize many single axis and group **MOVE**s so they all start simultaneously. This is useful when you have multiple axes with moves that are largely independent, but must start at the same time. It is also useful for coordinating a single axis with group.

Synchronization is controlled with <***axis***>**.STARTTYPE**, and **SYNCSTART**. The feature allows you to load the motion generator with a motion command, but delaying the generation of motion until **SYNCSTART** is issued.

For example:

```
Group1.StartType = SYNC
AuxAxis.StartType = SYNC
Move Group1 {500, 500} VCruise = 2000
Sleep 1000    'Program delayed between Move Commands
Move AuxAxis 100 VCruise = 1000
SyncStart Group1 AuxAxis
```

causes the two profiles to start at the same time.

<***axis***>**.STARTTYPE** is overridden inside move commands so the above example can be done more conveniently:

```
Move Group1 {500, 500} VCruise = 2000 StartType = SYNC
Move AuxAxis 100 VCruise = 1000 StartType = SYNC
SyncStart Group1 AuxAxis
```

You can synchronize as many groups and axes as you want, including simulated axes. Since each **SYNCSTART** specifies the groups and axes it synchronizes, you can independently synchronize multiple sets of groups and axes.

### 6.10.6. Clear A Pending Move

If you have loaded a synchronized move in the motion generator and need to delete it, use **SYNCCLEAR**. For example, if you entered the following command in the above sequence before issuing **SYNCSTART**:

```
SyncClear Group1
```

The Group1 move is deleted. Changing GROUP1.STARTTYPE only affects subsequent moves. You must use **SYNCCLEAR** to clear pending synchronized moves. **SYNCCLEAR** has no effect once the move is executing. In this case, you stop the move as you would a non-synchronized move.

## *6. 11 VELOCITY OVERRIDE*

The MC provides the ability to speed up or slow down all motion commands. This can be applied to an entire machine at once, an entire group, or to individual axes independently. This capability is used extensively in machine development. You can adjust the entire machine speed with a single command. Because the command can be issued from the terminal, you can observe machine operation at a variety of speeds without modifying your program.

All axes on the MC are controlled with **SYSTEM.VELOCITYOVERRIDE** (**SYS.VORD**). For example:

```
Sys.VOSpd = 25
```

immediately (if the velocity is previously greater than 25) reduces the velocity of all currently executing **MOVE**s, **CIRCLE**s and **JOG**s, as well as any subsequently issued commands. Since, the velocities are controlled by **VOSPD**, the acceleration and jerk rates of all axes are proportionally adjusted. The final positions of **MOVE**s are not affected.

Use <***group***>**.VELOCITYOVERRIDE** (<***group***>**.VORD**) to override the velocity of a group rather than the entire machine. It is used similarly to **SYS.VORD**, except it applies only to the group. **VORD** can be applied to as many or as few groups as is desired and the amount of override specified for one axis is independent of the others.

If you use **SYS.VORD** and <***group***>**.VORD** simultaneously, the axis speed is reduced by the product of both override properties. For example:

```
Sys.VOrd = 66           'Reduce entire system to 2/3 speed
GroupB.VOrd = 50        'Reduce GroupB to 1/3 speed
```

You cannot override an axis in a group. Once an axis is part of a group, it cannot be controlled independently.

### 6.11.1.1. MOTION ASSIGNMENT

After assignment, a generic element can be used in all motion commands as a representative of another, usually "real", motion element. All activities performed on the generic element identically affect the "real" element.

```
Common Shared Scara As Group AxNm = A1 AxNm = A2 AxNm = A3 AxNm = A4 Model = 4
Gen_Group = Scara
In Task1.Prg:
' Scara is attached to Task1.Prg through its generic representative Gen_Group
Attach Gen_Group
  ? "Scara is attached to " Scara.AttachedTo
  ? "Initial position of Scara is " Scara.PFb
' Scara is moved through its generic representative Gen_Group
  Move Gen_Group {-300,-500,-40,0} absolute = 1
  Sleep 200
  ? "Final position of Robot is " Scara.PFb
  …
Detach Gen_Group
-> Scara is attached to TASK1.PRG
-> Initial position of Scara is {0 , 0 , 0 , 0}
-> Final position of Scara is {-300 , -500 , -40 , 0}
```

### 6.11.1.2. MOTION PROPERTIES

Through assignment, a generic element acquires all properties of the element being assigned to it. Properties of "real" elements can be queried through their generic representatives.

```
Gen_Axes_List[1] = A1
While Gen_Axes_List[1].IsMoving <> 0
   Sleep 1
End While
Sys.Dout.1 = 1
```

Changes made in properties of generic elements identically affect the properties in the referenced "real" elements.

```
Print "Position Factor and Acceleration of A1 Before SetUp: " A1.Pfac, A1.Acc
' Set A1's properties through Gen_Axes_List[1]
Call AxisSetUp(Gen_Axes_List[1])
Print "Position Factor and Acceleration of A1 After SetUp: " A1.Pfac, A1.Acc
-> Position Factor and Acceleration of A1 Before SetUp: 65536   1000
-> Position Factor and Acceleration of A1 After SetUp:  32768  1500
```

# 7. COMPENSATION TABLES

Groups allow you to control multiple axes as a single mechanism. With groups, the position command and feedback signals are no longer single values, but instead, are vectors with two or three elements. Velocity and acceleration no longer apply to a motor, but instead, apply to the combination of two or three motors moving in concert.

Some applications require high accuracy in position, so compensate for mechanical inaccuracies in the system is required. The MC allows you to define a table of corrections for one or more axes. This table defines a correction to the positioned command given to the axes, depending on its location or other axes' locations. This correction is achieved every SERCOS cycle. The compensation table is usually created by high accuracy tools and is then used by the MC to correct small inaccuracies. The correction can also take affect when the target axis is part of a group.

## 7.1.1. Specification

The compensation table is constructed from *N* axes, for which compensation is defined. The number of rows in the table will be k1*k2*….*k*n*, where k1 through k*n* are the number of correction points defined for the axes A1 through A*n,* respectively. The "source" positions (in source axis position units) must be equally spaced and monotonously increasing, while the "compensations" (in target user position units) are the corrections added to the "target" axes.

A compensation table 3 axes source on 3 axes target is shown below.

| **The Header** | | | | | |
|---|---|---|---|---|---|
| | | **3** | **10** | **2** | |
| **"Source" axis location** | | | **"Target" axis compensation** | | |
| 1 | 3.1 | 4.5 | 0 | 0.5 | 0.9 |
| 2 | 3.1 | 4.5 | 0.9 | 0.51 | 0.2 |
| --- | --- | --- | --- | --- | --- |
| 6 | 3.1 | 4.5 | 2.0 | 0.5 | 0.9 |
| 1 | 3.2 | 4.5 | 0 | 0.1 | 0.9 |
| 2 | 3.2 | 4.5 | 0.65 | 0.01 | 0.9 |
| --- | --- | --- | --- | --- | --- |
| 6 | 3.2 | 4.5 | 3.0 | 0.22 | 0.9 |
| --- | --- | --- | --- | --- | --- |
| 6 | 4 | 4.5 | 2.0 | 0.2 | 0.9 |
| 1 | 3.1 | 5.0 | 0 | 0.1 | 0.87 |
| --- | --- | --- | --- | --- | --- |
| 6 | 4 | 5.0 | 1 | 1 | 1.2 |

The value in the table is locked and calculated according to the source axis' **PCMD** and is linearly transformed. The "source" axes determine the correction value, depending on location. The "target" axes are corrected.

The table has a "header" which consists of the number of axes in the table followed by the number of compensation points defined for each axis. The table can be created the MC using **TARGETDATA**, **SOURCEDATA**, and **CREATECOMP**, or can be loaded from a binary file with a .CMP extension.

The correction takes affect when ALL the "source" axis position feedbacks reach the first (i.e., the minimal) value in the table. The target axes are added and the value calculated in the table to its current **PCMD** and **PFB**.

The values in the “source” columns are in absolute units, while the “target” values are in relative units (i.e., the value specified in the table is added to the current position of the axis, whether it is a geared, cammed or a regular axis). A value different from 0 at the beginning of the table causes a jump at the target axes whenever the source axes get to the minimum value specified in the table.

The difference between the corrected position (**COMPPCMD**) and the actual position feedback determines the position error of the target axes. **PCMD** returns the position of the generator before correction.

The source PFB (for example, source1.pcmd=1, source2.pcmd=3.2, source3.pcmd=4.5) is searched in the table and linearly transformed from the previous points in the table to give the correction for the target positions.

While the source axis continues its movement along the specified zone, the target axes change their positions according to the table. When one of the sources goes beyond the max value there are no more correction added for any of the axes in the table (correction=0). A jump may occur.

Correction values are assumed very small and will not cause a large jump in the target axis. **VOSPD** and **PEMAX** are affected by the corrected position. Change these values accordingly to avoid exceeding the maximum values.

While the compensation table is active, the values of the table cannot be changed. After the table is created, only the “target” values can be change.

## 7.1.2. Access Data

The compensation table data is arranged in the form of following target data for the points for each individual axis. If there are three axes (A1, A3, A3), there are three (*N*1, *N*2, *N*3) compensation points defined. To calculate the index to access the table with the indexes i, j, k for A1,A2,A3, use:

```
index = i + j*N1 + k*N1*N2 .
```

## 7.1.3. Define

The compensation table is defined as a global variable.

```
Common shared comp1 as comp
CreateComp  Comp1 3 ,4, 8
‘Creates a table with 3*4*8 number of rows and
'6 number of columns – 3 for the source & 3 for the target.
CompSet Comp1 A1 ,A3, A4 on A2, A6, A5
‘Sources A1 A3 A4 and targets A2 A6 A5 respectively.
```

The maximum and minimum positions of the table are set for every axis in the compensation table as well as a multiplier:

```
Comp1.minposition[2]  = 1.32
Comp1.maxposition[2] = 4.5
' Set the maximum value for the second axis in the table
```

## 7.1.4. Load/Save From a File

Another way to create a compensation table is to directly load it from a binary file stored on the Flash disk. The structure of the compensation file is:

**Header** – *<number of axes> <number of points axis1>* ….<number of points axisn> (integer 4 bytes)

**Source data** – *<min axis1> <max axis1>*…..*<min axis*n><*max axis*n> (double 8 bytes)

**Target data** – target values for axis1,…….target values for axis*n*. (double 8 bytes)

The file can be created directly from Basic Moves by loading a list of values (arranged as described above) in a .CSV format into the MC's Flash as a binary file with the CMP extension (see **LOADCOMPDATA** and **STORECOMPDATA** in the *SERVO**STAR**® MC Referenc*e manual).

## 7.1.5. Set and Query Values

While the comensation table is not active, you can change a specific target value.

```
Comp1.targetdata[1][23] = 3
‘ Set a new compensation value of index 23 of Axis1
?Comp1.sourcedata[2][24]
‘ Read the source value index 23 of Axis1
```

## 7.1.6. Activate

After defining the correction table turn it ON/OFF with **COMPACTIVE**. Before activating, a check is performed to insure that the table is valid (the points are evenly spaced and increasing). After validation and enabling the target axis, positions of the target axes are corrected using the value calculated from the compensation table. If a target axis is disabled, the correction takes effect only after the axis is enabled.

**NOTE**

***A jump may occur when the compensation procedure starts from a point other than the start point of the table. A jump may also occur if the compensation process is disabled before reaching the end of the table.***

## 7.1.7. Query Actual Positions

The actual positions are return by **COMPPCMD** and **COMPPFB**. These properties can be operated as the master source for a slave – geared or cammed. **PCMD** and **PFB** return the command and feedback positions *without* compensation.

## 7.1.8. Multi-Dimensional Correction

In the following example, the correction is performed on several axes. In the described system, there are three axes (X, Y, Z). The X, Y, Z position corrections are determined according to the location of the X, Y, and Z axes.

The table is composed from 4, 9, 3 entries for X, Y, Z axes, respectively.

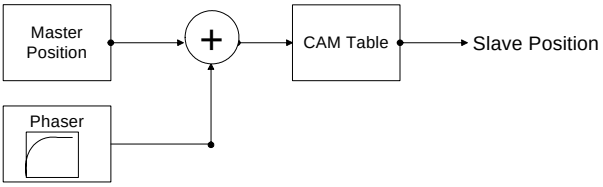
```
Common shared comp1 as comp                      'This declares a table in the system.
CreateComp comp1 4 ,9, 3
Comp1.minposition[1] = 0.1                       'Setting min for the X
Comp1.maxposition[1] = 9                         'Setting max for the X
Comp1.minposition[2] = 1.1                       'Setting min for the Y
Comp1.maxposition[2] = 2                         'Setting max for the Y
Comp1.minposition[3] = 3                         'Setting min for the Z
Comp1.maxposition[3] = 3.12                      'Setting max for the Z
CompSet comp1 ax1 ,ax2, ax3 on ax1 ,ax2, ax3     'Declare before target data definition
Comp1.TargetData[1][1] = 0.01                    'Setting the target
Comp1.TargetData[3][3*9*4] = 0.02                'Setting the target
Comp1.CompActive=1
```

From now on, any movement on any of the axes in the table generates compensation.

# 8. PHASER

The purpose of **PHASER** is to amplify or attenuate the master input value before entering the cam table or before multiplying by the gear ratio.

The master - slave connection is achieved either by a cam table or through gearing. A typical problem in camming is an offset in the master position that has to be compensated for before translating the value with a cam table. **PHASER** adds a correction to the master position. This avoids unexpected jumps at the slave position. By immediately adding the maximum correction value, there is a dynamically computation of the correction from zero to the maximum value.




Several slaves can be connected to the same master, but the position correction is set per slave. **PHASER** is a slave command and its effects do not harm other slaves connected to the same master. Directly disconnection of the slave (SLAVE = OFF) automatically stops **PHASER**.

## 8.1.1. Profiler

**PHASER** is a type of **MOVE**, so all the command's rules and limitations are imported. As an input, **PHASER** gets the maximum offset added or subtracted from the master position. The offset value is equivalent to the target position of **MOVE**. The rate at which the offset changes is determines by the profiler. Similar to **MOVE**, the profiler is calculated according to the slave's motion properties (**ACCELERATION**, **DECELERATION** and **JERK**). All changes in these values must be performed in the nodal context so the slave's property definitions are not effected.

Comparing **PHASER** and **MOVE**:

| Properties | MOVE | PHASER |
|---|---|---|
| Element | Axis or group | Slaved axis only |
| Position | Target | Maximum offset |
| Ratio factor | None | Translation factor between master units and slave units |
| Kinematics properties | All | All |
| **STARTTYPE** | All types | All types (special relevance is IMMEDIATE) |

## 8.1.2. Execute

**PHASER** is a slave command, although it acts on the master signal. In the command (like **MOVE**), the slave axis and offset value are essential. With **PHASER**, there is a new factor ratio that translates between slave and master units. The additional properties that can be determined are the smoothness of change, profiler type, **STARTTYPE** and kinematics.

```
A1.mastersource = a2.pcmd
A1.slave = CAM
Phase a1 10 ratio = 1  ‘ slave and master has the same units
```

## 8.1.3. Cancel

**PHASER** can be divided into two stages: the convergence and constant stages. Stopping the convergence stage is by a special stop command. The value reached at the end of stopping is the maximum offset that is added, instead of the original offset value. The maximum offset value is queried with **MAXIMUMOFFSET** (see the *SERVO**STAR**® MC. Reference manual*).

To cancel the second stage, where the offset reaches its final value, insert a new **PHASER** with a complementary correction value.

```
StopPhase a1 ratio = 1 ‘ slave and master has the same units
```

## 8.1.4. Serial **PHASER**

**PHASER** is inserted into the movements' buffer. During the convergence stage of **PHASER**, no other movements are allowed. A new movement or an additional **PHASER** is held until the maximum offset is reached. Executing a new motion during the convergence stage can be accomplished only with **STARTTYPE**=IMMEDIATE. You can trace the convergence stage with **ISMOVING** (like tracing other type of movements (see the *SERVO**STAR**® MC. Reference manual*).

# 9. GENERIC ELEMENTS

Writing complex multi-axis applications requires high level software organization. Avoid code repetition by encapsulating blocks of code into functions or subroutines. Code blocks of setup or query properties, which can be applied for all (or at least several) groups or axes in the system, and can be assembled into functions and subroutines. For example, if there are a number of identical motors in the system, their setup is the same. Write the following lines only once, instead of repeating them for each axis:

```
A1.PFac = 0x8000
A1.VFac = A1.PFac/1000
A1.AFac = A1.VFac/1000
A1.JFac = A1.AFac/1000
A1.VMax = 290
A1.AMax = 1500
A1.DMax = A1.AMax
A1.Acc = A1.Amax
A1.Dec = A1.DMax
…
```

Another example is writing common procedures as subroutines or functions that can later be applied in any system independently of how many axes or groups are defined. For example, switching on an output when an axis reaches its target:

```
While A1.IsMoving <> 0
   Sleep 1
End While
Sys.Dout.1 = 1
```

## *9. 1 ELEMENTID*

Depending on the system configuration, the MC can have several axes or groups defined. These motion elements are actually user interfaces of the system's internal memory or internal function calls. They are pointers to actual groups and axes.

During system configuration (“Sys.NumberAxes = *n*” and group definitions), each axis and group receives a unique element identifier. Axes get successive element identifiers ranging from 1 to 32, according to axis number. Groups get successive identifiers ranging from 33 to 64, according to declaration order. You cannot change these element identifiers.

The value of the element identifier is queried directly with **ELEMENTID** (read-only).

```
SYS.NUMBERAXES = 4
Common Shared G2 As Group AxNm = A3 AxNm = A4
Common Shared G1 As Group AxNm = A1 AxNm = A2

? A1.ELEMENTID
1
? A2.ELEMENTID
2
? G2.ELEMENTID
33
? G1.ELEMENTID
34
```

## 9.1.1. Declaration

Generic elements resemble other MC-Basic variables (longs, doubles, strings) in declaration syntax and scope (global, static and local), as well as in the ability to define arrays of up to 10 dimensions.

```
COMMON SHARED|DIM {SHARED} <axis_name> {[]…} AS GENERIC AXIS
COMMON SHARED|DIM {SHARED} <group_name> {[]…} AS GENERIC GROUP
```

## 9.1.2. Assignment

```
<generic_element_name>{[]…} = <real_element_name>
<generic_element_name>{[]…} = <generic_element_name>{[]…}
```

During declaration, all generic elements receive a zero element identifier, which prevents their usage as motion elements prior to assignment. If an unassigned generic element is used, an error is returned:

```
Common Shared Gen_Axis As Generic Axis

? Gen_Axis.ElementID
0
? Gen_Axis.VMax
Error:  3005, "Nonexistent group or axis", Module: Motion
```

Through assignment, a generic element receives the element identifier of another motion element, either “real” or generic, thus becoming a pointer to this element and acquiring all its properties.

```
Gen_Axis = A1
? Gen_Axis.ElementID
1
? A1.VMax
290
? Gen_Axis.VMax
290
```

By recurring assignments, generic elements have the ability to change their pointed axis or group as many times as desired.

```
Gen_Axis = A2
? Gen_Axis.ElementID
2
? A2.VMax
360
? Gen_Axis.VMax
360
```

The left-side (assigned) element of the assignment statement cannot be a “real” axis or group. It must always be a generic element.

```
A1 = Gen_Axis
Error:  7039, "Syntax Error", Module: Translator
```

The right-side element of the assignment statement can be either generic or a “real”. A group cannot be assigned to a generic axis, and an axis cannot assign a generic group.

```
Common Shared Gen_Group As Generic Group

Gen_Axis = G1
Error:  7067, "Wrong input type", Module: Translator
Gen_Group = A1
Error:  7067, "Wrong input type", Module: Translator
```

Joint axes cannot be used in either sides of assignment statement.

```
Gen_Group.j1 = A1
Error:  7039, "Syntax Error", Module: Translator
Gen_Axis = G1.j1
Error:  7039, "Syntax Error", Module: Translator
```

## 9.1.3. Limitations

Generic elements cannot be printed. Arithmetic, logic and bitwise operators cannot be applied on generic elements.

Generic elements cannot be used as conditions in flow control statements and event definitions.

Generic elements cannot be recorded.

Generic elements cannot be structure elements.

Generic element cannot serve as parameters of C functions.

Deletion of global generic element cannot be performed with **DELETEVAR** or **DELETEGROUP**. Hardware or software reset (**RESET ALL**) is required.

```
Dim Shared Gen_Axes_List[32] As Generic Axis
Gen_Axes_List[1] = A1
Gen_Axes_List[2] = A2
Gen_Axes_List[3] = A3
Gen_Axes_List[4] = A4
? Gen_Axes__List[1]
Error:  7039, "Syntax Error", Module: Translator
Gen_Axes__List[4] = Gen_Axes__List[1]+ 3
Error:  7039, "Syntax Error", Module: Translator
If Gen_Axes__List[3] > Gen_Axes__List[2] Then
   → Syntax Error
DeleteGroup Gen_Group
‘ Gen_Group is a global generic group
Error:  7039, "Syntax Error", Module: Translator
```

## 9.1.4. Functions

Generic elements can be defined “by reference” in MC-Basic functions and subroutines. Axes or groups passed as arguments could be either “real” or generic. For example, axis setup can be encapsulated in a subroutine. In this example, the subroutine arguments are “real” axes:

```
Sub AxisSetUp(Ax As Generic Axis)
   Ax.PFac = 0x8000
   Ax.VFac = Ax.PFac/1000
   Ax.AFac = Ax.VFac/1000
   Ax.JFac = Ax.AFac/1000
   Ax.VMax = 290
   Ax.AMax = 1500
   Ax.DMax = Ax.Amax
   Ax.Acc = Ax.Amax
   Ax.Dec = Ax.DMax
   …
End Sub
Call AxisSetUp(A1)
Call AxisSetUp(A2)
Call AxisSetUp(A3)
Call AxisSetUp(A4)
```

An entire array of generic elements can be passed (by reference) to functions and subroutines. Here, the argument is generic:

```
Sub AxesListSetUp(AxList[*] As Generic Axis)
   Dim i as long
   For i = 1 To Sys.NumberAxes
      AxList[i].PFac = 0x8000
      AxList[i].VFac = AxList[i].PFac/1000
      AxList[i].AFac = AxList[i].VFac/1000
      AxList[i].JFac = AxList[i].AFac/1000
      …
   Next
End Sub
Call AxesListSetUp(Gen_Axes_List)
```

The element type (i.e., axis or group) of an argument must match the function or subroutine declaration. Otherwise, a translation error occurs.

```
Call AxisSetUp(G1)                ‘ G1 is a group
-> Variable passed by reference has another type as in subroutine/function
declaration
```

Trying to define generic elements “by value” generates an error:

```
Sub AxisSetUp(ByVal Ax As Generic Axis)
   …
End Sub
-> Cannot pass an axis or a group by value to subroutine/function
```

Joint (J) axes cannot serve as function or subroutine arguments.

```
Call AxisSetUp(G1.j1)
-> Syntax Error
```

Generic elements can also serve as returned values of MC-Basic functions. Both “real” and generic elements can be assigned to returned values. For example, to generalize the former code, an axis-indexing function can be added, in which a “real” axis is assigned to the return value:

```
Function JointAxisByIndex(Byval I As Long) As Generic Axis
   Select Case I
      Case 1
          JointAxisByIndex = G1.J1
      Case 2
          JointAxisByIndex = G1.J2
      …
End Select
End Function
```

Then, write a general loop:

```
Dim A As Generic Axis
For Index = 1 to Sys.NumberAxes
A = AxisByIndex(Index)
   Call AxisSetUp(A)
Next
```

The generic element-returning function itself cannot be used as an argument since generic elements can only be passed by reference,and a function call is passed by value.

```
For Index = 1 to Sys.NumberAxes
   Call AxisSetUp( AxisByIndex(Index) )
Next
-> Cannot pass a constant or a complex expression by reference to
subroutine/function
```

Joint (J) axes cannot be assigned to return values.

```
Function JointAxisByIndex(Byval I As Long) As Generic Axis
   Select Case I
      Case 1
          JointAxisByIndex = G1.J1   →   Syntax Error
      Case 2
          JointAxisByIndex = G1.J2   →   Syntax Error
      …
End Select
End Function
```

# 9. 2 WITH

**WITH** can be given in three contexts: configuration file, terminal and task. All three contexts of **WITH** can be applied to generic elements.

```
With Gen_Axis
PFac = 0x8000
   VFac = PFac/1000
   AFac = VFac/1000
   JFac = AFac/1000
   …
End With
```

**WITH** can be used on assigned and unassigned generic elements. These generic elements can be assigned and/or re-assigned later. This changes the **WITH** element without writing a new **WITH** command.

```
With Gen_Axis
? With
GEN_AXIS
? VMax
Error:  3005, "Nonexistent group or axis", Module: Motion
Gen_Axis = A1
? VMax
290
Gen_Axis = A2
? VMax
360
```

# 10. INPUT/OUTPUT

The MC provides numerous types of I/O. It includes more than 40 I/O points as standard plus the SERVO**STAR** drive includes 5 I/O points per drive. In addition, you can input an external encoder through any SERVO**STAR** drive. The I/O can be extended to include PC104 cards from a variety of suppliers.

## *10. 1 STANDARD I/O*

The MC includes 23 inputs and 20 outputs as standard on the MC. These are accessed with the system properties DIN.1 through DIN.23 and DOUT.1 through DOUT.20:

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

For information concerning the properties of the signals that connect to these inputs and outputs, refer to the *SERVO**STAR**® MC Installatiion Manual.*

## *10. 2 SOFTWARE I/O*

The MC provides virtual software I/O in addition to the standard I/O. The primary purpose of software I/O is to place a large group of I/O where it can be read from and written to as blocks in the MC's dual port RAM. Virtual inputs are delivered from the host over DPRAM where virtual outputs are stored in the DPRAM and retrieved by the host. This is much faster than reading or writing individual I/O points one at a time. This allows efficient communication between a machine controller, such as a VB program or a software PLC, and the MC program. For more information about virtual software I/O, refer to the *SERVO**STAR**® MC API Manual.*

### 10.2.1. Bit-oriented Software I/O

Software I/O is used in the MC program just like standard I/O. Standard inputs are accessed with SYSTEM.DIN.1 through DIN.23, and virtual inputs are accessed with SYSTEM.VIN.1 through VIN.32. Standard outputs are accessed with SYSTEM.DOUT.1 through DOUT.20, and virtual outputs are accessed with SYSTEM.VOUT.1 through VOUT.32. You can use software I/O to start events just as with standard I/O. Programmable limit switches can use software outputs just as with standard outputs. Here is an example:

```
OnEvent SoftwareInputEvent System.VIn.1 = ON
System.VOut.1 = OFF
If(System.VOut.4 = ON)
 . . .
```

## 10.2.2. Long-Word-Oriented Software I/O

The entire word (32 bits) of software input and output is accessed when a bit position is not included in the reference to the input or output variable. So the following statement is valid:

```
System.VOut=System.VIn
```

This statement copies the virtual software input to the output. The physical input and output variables are used in this manner. Be careful when assigning virtual I/O to physical I/O because there are fewer physical bits than there are virtual bits. There are only 20 bits of physical output, but the virtual output variable can have 32 bits.

# *10. 3 PLS (PROGRAMMABLE LIMIT SWITCH)*

A Programmable Limit Switch (PLS) monitors position feedback on any axis, real or simulated. A PLS offers more flexibility than a limit switch, as it allows many trip points and 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 that allows you to change individual values of PLS positions. This array must be increasing monotonic. There is no limit of the number of positions or PLSs that can be defined.

The initial output polarity is specified with **PLSPOLARITY**. The output state is set when the PLS is enabled.

## 10.3.1. Enable and Disable

After being defined, a PLS is disabled (output pattern is not generated). The output is set when the PLS is enabled. The output is set to the defined initial state.

PLS properties are changed only when the PLS is disabled. Each enabled PLS requires CPU resources. Conserve those resources by disabling the unused PLS(s). **PLSENABLE** controls the status of the PLS:

```
MyPLS.PlsEnable = ON | OFF
```

**PLSENABLE** queries the status of a PLS:

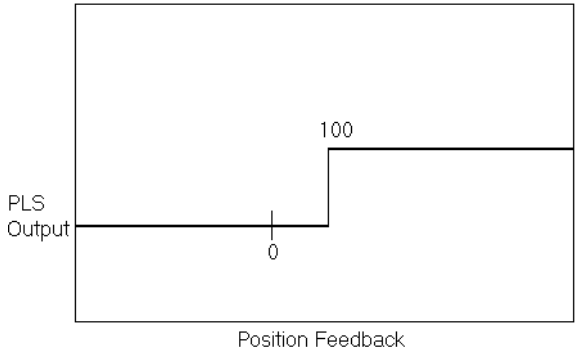
```
?MyPls.PlsEnable
if(MyPLS.PlsEnable = OFF)
 . . .
```

or drives events:

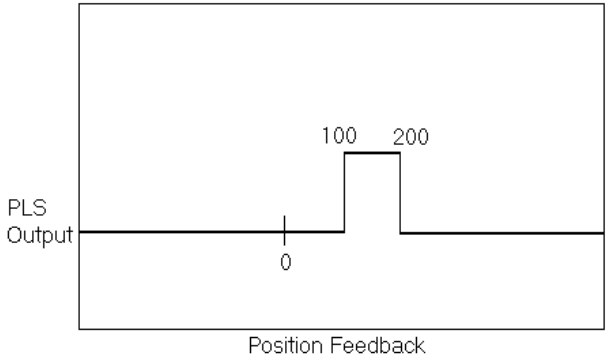
```
EventOn MyEvent MyPLS.PlsOutput = ON
```

## 10.3.2. Switch Positions

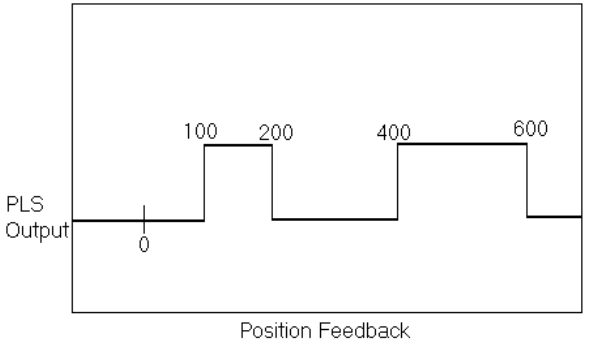
The basic mechanism of the PLS is the switch position(s) where the PLS output changes state. The figure belows shows a PLS output with one switch position at 100.




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



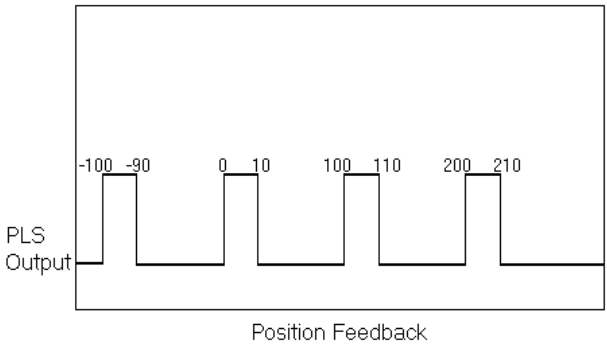

You can add any number of positions. For example, to extend the above example to turn back on at 400 and off again at 600, add positions 3 and 4 as 400 and 600 as shown below:



There is no explicit limit to the number of positions you can have in a PLS, but the PLS positions must be increasing monotonic (Position2 must be greater than Position1, Position3 must be greater than Position2, etc.).

## 10.3.3. Repetition Interval

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




## 10.3.4. Polarity

**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 is 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 is set to 0 or 1, according to the current position so that when the axis reaches the first segment, the output polarity is as indicated. The default value for polarity is **0**.

## 10.3.5. Hysteresis

The MC allows you to apply hysteresis around switch points. Hysteresis provides a margin in the switching between PLS positions to accommodate noise in the PLS axis. 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 0.01 position units, each transition of PLS position is moved forward 0.005 units for positive motion and backward 0.005 units for negative motion. 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 does not change due to this noise.

## 10.3.6. Enable and Disable

**PLSENABLE** enables or disables the PLS(s). After being defined, a PLS is disabled. The output is fixed when the PLS is enabled. The PLS output is set to the defined initial state when the PLS is enabled.

## 10.3.7. PLS Output State

Query the digital output associated with the PLS to query the state of the PLS output. For example, assume the PLS output is assigned to SYSTEM.DOUT.1. To determine the output state of the PLS, query the state of SYSTEM.DOUT.1:

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

## 10.3.8. Set Up

### Step 1

Declare the PLS with **Common Shared** <***var***> **as PLS** in Config.Prg or at the terminal. You must name the PLS, specify the axis that drives the PLS, and name an output to be controlled by the PLS. For example:

```
Common Shared MyPLS as PLS A1 System.Dout.1
```

sets up MyPLS as a PLS driven by axis A1 connected to SYSTEM.DOUT.1. The output can be a digital output (SYSTEM.DOUT.1 through DOUT.20) or it can be one of the virtual outputs (SYSTEM.VOUT.1 through VOUT.32).

After the **Common Shared** <***var***> **as PLS** executes, 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 using **PLSOUTPUT**.

### Step 2

Create the PLS data structure and define PLS positions. The number of defined positions is not explicitly limited. A call to **CREATEPLSDATA** creates an array of *n* positions that stores the position data. The array is 1-based and an error is 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. Position values must be increasing monotonically.

### Step 3

Set the polarity. **PLSPOLARITY** defaults to OFF, indicating the PLS state should be OFF after the first position. Change **PLSPOLARITY** to ON to invert the state.

```
MyPLS.PLSPolarity = ON
```

Change **PLSPOLARITY** only if the PLS is disabled.

### Step 4

Set **PLSREPEAT**. If you want the PLS to repeat, set the value of **PLSREPEAT** to a non-zero, positive number. For example, to have the PLS repeat every 10,000 position units, enter:

```
MyPLS.PLSRepeat = 10000
```

**PLSREPEAT** defaults to 0, indicating that there is no repetition. Change **PLSREPEAT** only when the PLS is disabled.

### Step 5

Set **PLSHYSTERESIS**, *if the application requires it*. Hysteresis is only necessary if the system does 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.PLSHysteresis = 0.01
```

### Step 6

Enable the PLS. Enter:

```
MyPLS.PLSEnable = ON
```

There are a few other PLS functions you may need:

**Change Polarity** The default output state of a PLS is 0. Change this initial state by modifying **PLSPOLARITY**.

```
MyPLS.PLSPolarity = 1
```

The PLS must be disabled when changing this setting.

**Query the Name** Query the name of the axis driving the PLS:

```
? MyPLS.PLSAxisName
```

returns the axis associated with the PLS (A1).

**Disable the PLS** Disable the PLS. For example:

```
MyPLS.PLSEnable = OFF
```

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

**Delete a PLS** Delete a PLS to remove it from the system only when the PLS is disable and there are no tasks in memory. For example:

```
DeletePLS MyPLS
```

## *10. 4 EXTERNAL ENCODERS*

The MC supports external encoders (encoders not connected to an axis controlled by the MC). These encoders are connected to the system using the external encoder input of any SERVO**STAR** drive in the system. You can access the encoder periodically with:

```
? IDNValue(<Axis>.DriveAddress, 53, 7)
‘Query external encoder position
```

If you are using the encoder for master/slave operation or in a way that requires the value of the signal be updated to the MC every SERCOS cycle, you must configure the telegram for that axis as Telegram type 7. You also must specify that external position must be included in the telegram. This command is only available when in communication phase of 2 or higher.

# *10. 5 PC104*

If you need more I/O than the 23 digital inputs and 20 digital outputs that the MC provides, there is an optional PC104 bus. This lets you install PC104-compatible cards to provide additional I/O ports. Because the PC104 bus is an extension for the MC, this configuration is often called a mezzanine bus.

The MC supports the full PC104 mechanical specification for as many as two PC104 cards. Adding these cards generally increases, by one, the number of slots taken by the MC. The MC supports a subset of the electrical connections of PC104, but the subset is sufficient to accommodate nearly all PC104 I/O cards. Refer to the *SERVO**STAR**® MC Installation Manual* for more details concerning which pins are supported.

## 10.5.1. Configuring PC104 Cards

You must configure the PC104 cards for memory and I/O space. See the manufacturer's manual for your particular PC104 carde.

All PC104 cards used on the MC which are memory mapped, must be mapped into the MC's memory without causing conflict. If you are using two PC104 cards, you must be careful not to overlap the memory spaces of these cards.

All PC104 cards used on the MC which are I/O mapped, must be mapped between 0x100 to 0xFFFF. Address ranges: 0x300 through 0x3ff and 0x2f8 through 0x2ff are not available. If you are using two PC104 cards, you must be careful not to overlap the I/O spaces of these cards. ***Do not use ranges 0x300 through 0x3ff and 0x2f8 through 0x2ff, even though they are not protected by the software.***

***The MC does not support interrupts from PC104 cards.***

## 10.5.2. Installation

To install your PC104 card:

1. Power down your unit.
2. Always use proper static handling procedures.
3. Mount the cards.
4. Install stand-offs.

# 10.5.3. Commands

The MC supports PC104 with memory and I/O instructions. For further information on these commands, refer to the *SERVO**STAR**® MC Reference Manual.*

If you are using memory-mapped PC104 cards, you will need the memory access commands, **PEEK** and **POKE**. The valid memory address range for **PEEK** is 0xA0000 through 0xDFFFF. The valid memory address range for **POKE** depends on the hardware version of the MC card.

***For the ISA-Bus card***, the valid memory address range for **POKE** is 0xA0000 to 0xCFFFF and 0xDA000 to 0xDFFFF.

***For the PCI card***, the valid memory address range for **POKE** is 0xA0000 to 0xC7FFF and 0xDB000 to 0xDFFFF.

**PEEKB** transfers byte-wide (8-bit) data from the PC104 memory bus to a variable.

**PEEKW** transfers word-wide (16-bit) data from the PC104 memory bus to a variable.

**POKEB** transfers byte-wide (8-bit) data to the PC104 memory bus.

**POKEW** transfers word-wide (16-bit) data to the PC104 memory bus.

If you are using I/O-mapped PC104 cards, you need **INP** and **OUT**.

**INPB** transfers byte-wide (8-bit) data from the PC104 I/O bus to a variable.

**INPW** transfers word-wide (16-bit) data from the PC104 I/O bus to a variable.

**OUTB** transfers byte-wide (8-bit) data to the PC104 I/O bus.

**OUTW** transfers word-wide (16-bit) data to the PC104 I/O bus.

# 11. ERROR HANDLING

Error handling is the method used to react to and process fatal and non-fatal errors that occur during the operation of the MC. You can handle non-fatal faults, but not fatal faults. Fatal faults cause a WatchDog timer to be triggered. All errors have corresponding default actions to be taken when they occur.

There are three contexts in which an error can occur and be handled: task, system, and terminal. The following definitions will help you while reading through this chapter:

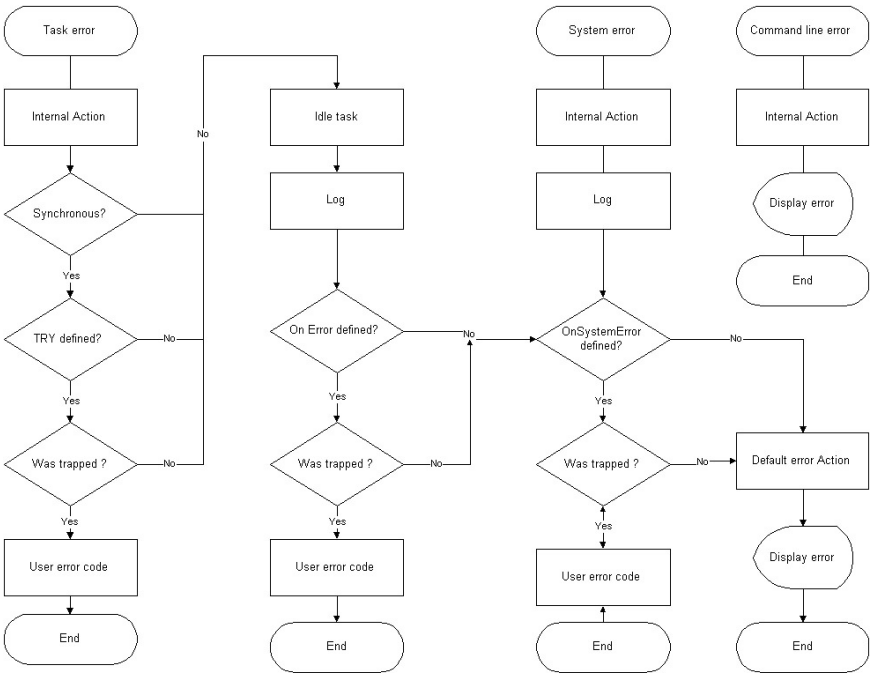
| Term | Definition |
|---|---|
| Internal error action | Immediate action taken by the system software when an error occurs. This action cannot be turned off or bypassed. |
| Default System error action | Action taken by the system according to the context and severity of the error if the user does not handle the error. All default system error actions can be bypassed 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 not associated with a specific line of program code. Examples include following error and motion overspeed. |

## *11. 1 CONTEXT*

When an error occurs in the MC system, it occurs in one of three contexts: task, system or terminal. It is important to recognize the differences among these three since the action taken by the system varies, depending on the context.

### 11.1.1. Task Context

Errors occurring as a result of an executing task take place in the task context. Asynchronous and synchronous errors occur in this context. A specific line of code causes 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. This 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 figure shows the process flow that occurs in the task context.




During the internal error action, the synchronous error is logged in the MC and the task is idled. The default system error handler stops all motion and all attached tasks.

In the figure above, 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.

## *11.1.1.1. ONERROR*

The **OnError** block is handled like an event handler. An event handler lets your program respond to events. The **OnError** block allows your program to catch errors. **OnError** overrides default system error action if **OnError** traps error. 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.

Add an **OnError** code block to your program between the **Program** and **End Program** section of your program.

The number of **Catch** statements in an **OnError** block is not explicitly limited. **Catch Else** may optionally be used. If you want a task to resume after an error has occurred, use **CONTINUETASK**.

You can use **GoTo** within the error handler. However, because **OnError** interrupts the main program, you cannot use **GoTo** to branch outside the error handler. You cannot place an **OnError** block in the middle of program flow. The scope of **OnError** is that of the task withing which it is contained. It only handlers errors that occur within that task.

#### 11.1.1.2. 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 only traps synchronous errors in the task context. The block is started using the **Try** keyword and is terminated with the **End Try** keyword.

**Try** blocks can be nested. The program lines between the line causing the error and the matching catch statement are skipped. The interpreter tries 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.

## 11.1.2. System Context

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 affect all running tasks. The default system error handler processes these errors. Examples of system context errors include Floating Point Unit errors, CPU errors, SERCOS communication errors and errors that occur on motion elements not attached to specific tasks.

#### 11.1.2.1. ONSYSTEMERROR

**OnSystemError** traps and processes all errors in the system context. It is the upper-level of the hierarchical 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 traps both synchronous and asynchronous errors in all tasks, as well as errors that occur within the context of the system.

A system error 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 not attached to a task. When an error is trapped, the specified error processing code runs and the task is stops. The task is in state 4. It is possible to continue task execution by explicitly entering **CONTINUETASK** within the error processing code, but the task continues only after the error has been corrected.

**OnSystemError** traps errors not specifically trapped by **Try** or by **OnError**. It then executes either an orderly shutdown of the system, or an orderly recovery procedure.

#### 11.1.2.2. ERRORPRINTLEVEL

**SYSTEM.ERRORPRINTLEVEL** controls which types of system errors are printed to the message log window. There are four levels of system errors: fatal faults, errors, and notes.

0 (SILENTLEVEL) – Notes, errors, and fatal faults are not printed.

1 (FAULTLEVEL) – Notes and errors are not printed. Fatal faults are printed.

2 (ERRORLEVEL) – Notes are not printed. Errors and fatal faults are printed.

3 (NOTELEVEL) – Notes, errors, and fatal faults are printed.

**SYSTEM.ERRORPRINTLEVEL** only affects printing to the message log window. Fatal faults and errors continue to be logged, and can be viewed using **?ERRORHISTORY**. **ERRORPRINTLEVEL** applies only to asynchronous errors. Synchronous errors are not affected.

### 11.1.3. Terminal Context

Terminal context is similar to task context, as commands are processed similarly. The error action is different from a task as there is no interpreter to be idled when an error occurs.

## *11. 2 WATCHDOG*

The MC provides a WatchDog timer to help determine if the host PC connection is operational. To use the WatchDog, configure the MC to expect WatchDog messages. During normal operation, the host PC must reset the WatchDog timer before the end of the specified interval to avoid a WatchDog fault.

Normally, the WatchDog is cycled indirectly through fast I/O. **WDCYCLE** directly resets the WatchDog, but this is not recommended because it requires 15 to 20 ms per WatchDog cycle. Given that WatchDog timers are often reset every 100 ms, this implies that the WatchDog function could consume 15 to 20% of the MC processor time. Events triggered by the fast I/O can avoid this problem. The following commands configure the MC to expect WatchDog cycles:

**WDINIT** establishes a WatchDog timer. When selecting the reset time, it should be long enough that the WatchDog function does not consume undue amounts of processing time. The time also should be short enough so that if the communication link between MC and host is lost, the MC can shut the axes down in a timely manner. Different applications require different times, but 100 to 1000 ms (<*Cycles*> = 5 to 50) are commonly used.

**WDCYCLE** cycles the WatchDog timer.

**WDDELETE** deletes the WatchDog timer. **WDDELETE** is seldom used in normal operation because once a WatchDog timer is started, it is seldom stopped intentionally.

## 11.2.1. WatchDog Setup

To set up the MC to enable the WatchDog, follow these steps:

1. Create a new task solely for the WatchDog. Use BASIC Moves with *File*, *New*.
2 Define a local variable as a *Long*. For the example below, we use *MyVar*.
3 Initialize the WatchDog timer in the main program. Store the return value of the initialization step in the local variable from Step 1.
4 Write an event inside the task to reset the WatchDog timer. The event is fired with fast I/O (SYSTEM.VIN.1 to 32).
5 Set the task to run infrequently so it does not consume more computational resources than necessary. The following example demonstrates a WatchDog timer task:

```
Dim Shared MyVar as Long
Program
'Event which runs once for each WatchDog cycle
OnEvent WDevent System.Vin.1 = 1 ScanTime = 4 'Check every 4th cycle
   WDCycle(MyVar) 'Cycle the WatchDog
   System.Vin.1 = 0                          'Reset the WatchDog bit
End OnEvent
   EventOn WDEvent                           'Enable the Wdog Event
   MyVar = WDInit(5)      'Set Wdog time for 5 Wdog cycles or 100 ms
   WDCycle(MyVar)            'You must cycle the Wdog once to begin
   While 1 = 1             'Infinitely loop so event will be active
      Sleep 1e6                      'Use sleep to conserve CPU time
   End While
End Program
```

With this example, the WatchDog must be reset every fifth cycle (every 100 ms) to avoid a fault.

## 11.2.2. Cycle the WatchDog

The last step in using the MC WatchDog is to issue watch cycle commands from the host PC. These are issued more frequently than the specified maximum cycle time in **WDINIT**. For example, if the maximum cycles are specified as 5 (equivalent to 100 ms), reset the WatchDog every 50 or 60 ms. In Visual Basic, this is as simple as setting the fast I/O to 1 based on a 50 ms timer.

## 11.2.3. Reset the WatchDog

You can reset the WatchDog directly from the host PC using **WDCYCLE** This is simpler than the method presented above because you need not generate a task. However, this is not normally recommended because it is much slower than the fast-I/O method as it requires 15 to 20 ms to pre-compile commands issued directly from the API to the MC. However, should you decide to use the direct method, you need only use the following steps directly from the API:

1. Create a global variable.
2. Initialize the WatchDog timer.

3. Reset the WatchDog with an API command based on a timer. Run the following Visual BASIC code once upon initialization:

```
Include KMAPI.BAS in your project
KMExecuteCmd(MC, "MyVar=WDInit(5)")
KMExecuteCmd(MC, "WDCycle(MyVar)")
```

4. Run the following Visual BASIC code from a timer set for 50 ms:

```
KMEXecuteEmd(MC, "WDCycle(MyVar)")
```

## *11. 3 UEA (USER ERROR ASSERTION)*

The purpose of **U**ser **E**rror **A**ssertion (UEA) is to let the application developer extend existing system (internal) errors. You can define application (additional) errors (exceptions) so the system handles them like it would its own. The application exception may be trapped with **TRY/Catch**, **OnError** or **OnSystemError**. Unprocessed application exceptions are treated as any known internal error. The system reacts by stopping the task/motion according to exception severity. The system provides a range for the application errors and prevents an overlap between internal and application error codes. The application is able to define an exception name and ASCII message to print. User exceptions start from number 20001 and you are able to define 1000 exceptions. UEA may have "Note" and "Error" seventies.

## *11. 4 EXCEPTIONS*

### 11.4.1. Declaration

The application developer can declare an exception and supply the corresponding error message. An exception may be declared on the both system and task levels. Declaration of exceptions at the subroutine level is not supported. User exceptions are divided into two sub ranges 20001-20499 and 20500-20999. Applications developed may use the first sub range for explicit definitions of exception number (see below example), while the second sub range is used by the system for automatic assignment of exception numbers. Double use of the same exception number is forbidden as the System gives an error. The exception can be defined with the following declaration:

```
Dim|common shared <name> as <severity> <ASCII message> [<num>]
```

where:

| Parameter | Description | Remarks |
|---|---|---|
| **Severity** | **Note/Error** | |
| Num (optional parameter) | Integer number 20001..20499, which will be assigned to exception. This parameter is optional, if omitted the system will assign some value in range 20500..20999. MC will report an error if number given by the application developer is already used. | |
| ACII Msg | ASCII text that will appear at error history and error message. | |

For example:

```
Dim shared Error1 as Note "Emergency" 20001
Dim shared Error2 as Note "Fault" 20002
Common shared MyNote as Note "My Note"
Common shared MyError as Error "My Error"
```

If the exception number is not specified, the system associates exceptions with some numeric value, which can be queried. However, the system will not guarantee that this value is the same from load to load. Do not make any assumption about this number. It is a mechanism to maintain the exception database.

## 11.4.2. Deletion

All exceptions declared at task level are lost when the task is killed. The exception declared at the system level (**Common Shared**) can be deleted if not in use – similar to numeric variables. For example:

```
DeleteVar MyNote
```

## 11.4.3. Assertion

The application asserts (throws) an exception from within an application. The system behaves like it is a regular (predefined in firmware) application error. You are able to catch application exception with **TRY/Catch**, **OnError** or **OnSystemError**.

Exception assertion is possible with **THROW**. **THROW** accepts the name of any defined exception. Other expressions are not allowed to be arguments to **THROW**.

The scope of **THROW** is not limited. It can be executed inside of **Try**, **Catch**, **Finally**, outside the Try block in **OnError** and **OnEvent**. For example:

```
Dim shared LimitError as Error msg= ″Over travel″
Program
      Attach a1
      Jog a1 100
      While 1
‘check the axis position and assert and error,
‘the system should stop the task and motion
         If a1.pfb > 1e10 Then
            Throw LimitError
         End If
      End while
End program
```

Another example:

```
Dim shared Error1 as Error msg= ″″          ‘ no message
Program
      Try
‘ check Input #1 and do not continue
‘ if it’s “1”
      If Sys.din.1=1 Then
         Throw Error1
      End if
      Move a1 1e3
   Catch Error1
      Print “I/O error”
   End try
End program
```

## 11.4.4. Log

User exceptions are printed and logged according to existing rules: A note is printed but not logged, Unhandled errors are printed and logge, etc. In addition, the application developer can log an error without any error handling. Application exception is printed and logged but neither task generated such an exception nor motion associated with that task stops.

User exceptions are logged according to the existing rules. Errors and faults are printed and logged while notes are only printed.

You can use **LOGGER** to lsend exceptions directly to the logger, bypassing any error handler. The exception can be logged with the following command:

```
Logger <Exception name>
```

For example:

```
Logger MyError
```

## 11.4.5. Query

You can query exception numbers and messages, but changing these parameters is forbidden. The exception number and text are set during exception definition and stay constant until the exception is deleted or the defined task is killed. For example:

```
→?MyError.num
20007
→?MyError.Msg
My Error
```

## 11.4.6. Print

Exceptions are printed and logged with timestamp, task name, and line number, severity and other error attributes. In contrast to internal (predefined) errors, an exception number is not constant and may vary from time to time (from task load to task load). Also, the error log may keep exceptions from tasks not in memory and there is no way to associate an exception number with its message. The error logger keeps exception messages in RAM and files.

## 11.4.7. Limitations

1. **VARLIST** and **SAVE** do not show exceptions.
2. **WATCH** is not supported.
3. Arrays of exceptions not supported.
4. Definition of exceptiosn in subroutines is not supported.
5. Arithmetical operations are not supported for exceptions. Comparing exceptions makes sense only if the application developer explicitly gives exception numbers. If the MC assigns exception numbers, the result is unpredictable.
6. The library can assert an exception defined as **DIM SHARED** or **COMMON SHARED**. Realize that exceptions defined in the library do not have constant numbers (if numbers were not assigned explicitly).

# APPENDIX A

## *SAMPLE NESTING PROGRAM*

```
Dim Shared I1 As Long
Dim Shared I2 As Long
Dim Shared I3 As Long
Dim Shared I4 As Long
Dim Shared I5 As Long
Program
   Do
      If I1=0 Then               'First If...Then
         If I2=0 Then            'Second If...Then
            While I3<5            'First While
               Print ""
               Print "*******I3=";
               ?I3
               I3=I3+1
               For I4=1 to I3 Step 2   'First For statement
                  Print "I4=";
                  ?I4
               For I5=I4 to 10      'Second For statement
                  Print "I5=";
                  ?I5
                  Sleep 50      'To slow task down
               Next I5
               Next I4
            End While
         End If
      Else
         I3=0
         While I3 <=5             'Second While
            I3=I3+1
            Select Case I3
               Case 1
                  Print "I3=1"
               Case 2
                  Print "I3=2"
               Case 3
                  Print "I3=3"
               Case Else
                  Print "I3 is not 1, 2, or 3"
            End Select
         End While
      End If
      I1=I1+1
   Loop Until I1>1
End Program
```

## 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
   colums=2
   array_z[1][1]=125.6

   REM specification of program
   <optional code>
      CALL OrdinarySubroutine(var_x,var_y)
   <more code>
      CALL MatrixMath(array_z[1][1],rows,columns)
   <more code>
End Program

SUB OrdinarySubroutine(param1 As Double, ByVal param2 As Long)
REM pass param1 by Reference, param2 by Value
   REM declare local variables if any
   REM specification of subroutine
   <optional code using param1 and param2>
End Sub

SUB MatrixMath(matrix As Double, ByVal rows As Long, ByVal columns 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
```

# SAMPLE AUTOSETUP PROGRAM

Below is an example auto setup program. This sample covers several pages. You do not need to understand the program in detail at this point but you should familiarize yourself with the structure.

***REMOVING THE SINGLE QUOTES IN THE FOLLOWING CODE CAUSES THE PROGRAM TO ENABLE THE DRIVES AND GENERATE MOTION COMMANDS! BE PREPARED FOR THE MACHINE TO MOVE!***

***ENABLING THE SYSTEM WITHOUT PROPER TUNING MAY CAUSE UNEXPECTED MOTION! BE CERTAIN YOUR SYSTEM IS TUNED PROPERLY! SEE THE SERVOSTAR® MC INSTALLATION MANUAL FOR MORE INFORMATION.***

```
rem ------------------------------------------------------------
rem TASK....: C:\Program Files\KMTG Motion Suite\BASIC
Moves\Projects\Junk2\Junk2.prg
rem AUTHOR..: <Name of Author Here>
rem TIME....: <Time of Creation Here>
rem PURPOSE.: Main Project File

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 A1                  'Attach to axes
   Call AxisSetup               'Set up all axes
   Call SercosSetup               'Bring up SERCOS ring
rem   Auto generated sample program starts here
rem   Sample program--remove single quote marks at beginning of
rem   lines to run program
'   Sys.En = on                  'Enable the system
'   A1.En = on                  'Enable Axis A1
'   A1.StartType = Immediate
'   Sleep 1000                  'Sleep one second
```

***Continued on next page…….***

```
'   Sys.Motion = on                  'Allow system motion
'   A1.Motion = on                   'Allow motion in axis A1
'   For sample_int = 1 To 10         'Loop 10 times

rem   Move A1 to 5 position unit(s)
'      Move A1 5 Absolute=0
'      Sleep 2000               'Sleep two seconds

rem   Move A1 back to start
'      Move A1 -5 Absolute=0
'      Sleep 2000               'Sleep two seconds
'   Next sample_int

'   Print "Sample program completed."

'   Sys.Motion = off               'Disable motion in system
'    Sys.En = off               'Disable system

rem   Auto generated sample program end here
   Detach A1                  'Detach from axes

End Program

rem ------------------------------------------------------------
rem   Sercos Set-up Routine Automatically Generated by the Project Wizard
rem ------------------------------------------------------------
Sub SercosSetup

rem   If ring is not up...
   If Sercos.Phase <> 4 Then
      Sercos.Phase = 0           'Bring the ring all the way down
      Sercos.Baudrate = 4        'Set baudrate to 4 Mbaud

rem   Set Drive Addresses
      A1.Dadd = 1

      Sercos.Phase = 4            'Bring ring all the way up
   End If
End Sub
```

***Continued on next page…….***

```
rem ----------------------------------------------------------------
rem Axis Set-up Routine Automatically Generated by Project Wizard
rem ----------------------------------------------------------------
Sub AxisSetup
   If Sercos.Phase = 4 then
      A1.enable = off                'make sure axis is disabled
   End If
   A1.PFac = 65536                    'Revolutions
   A1.VFac = A1.PFac / 1000           'Revolutions/second
   A1.AFac = A1.PFac / 1000 / 1000    'Revolutions/second/second
   A1.AMax = 10000
   A1.DMax = 10000
   A1.Acc = A1.AMax
   A1.Dec = A1.DMax
   A1.JMax=5*A1.DMax
   A1.Jerk=A1.JMax
   A1.VMax = 100
   A1.VOspd = A1.VMax * 1.2
   A1.VCruise = 50
   A1.PEMax = .01
   A1.Displacement = 0
   A1.StartType = GCom
   A1.Absolute = 1
   If Sercos.Phase = 0 then      'can only modify simulated
                                 'property in phase 0
      A1.Simulated = off
   End If
   If Sercos.Phase = 4 then
      A1.enable = on             'reenable axis
   End If

End Sub
SERCOS Setup Subroutine
The following example modifies the SERCOSSetup subroutine from the BASIC Moves
auto setup program:
Sub SercosSetup

 Rem   Setup Axis A1 for Telegram Type 7 with PExt in Cyclic Data
 Rem   Setup Axis 2 to Slave to Axis 1 PExt

 Rem  If the ring is up, skip this process
 If Sercos.Phase <> 4 Then

 Rem  Set Sercos.Phase to 0 and set the baud rate
   Sercos.Phase = 0
   Sercos.Baudrate = 4         'for 4 MBaud

   Rem   Set the drive addresses
   A1.DriveAddress = 1
   A2.DriveAddress = 2
   A1.Simulated = 1
   A2.Simulated = 2

   Rem   Set Sercos.Phase to 2
   Sercos.Phase = 2
```

***Continued on next page…….***

```
   Rem   Configure the axis for Telegram Type 7 using IDN 15
   WriteIDNValue Drive = A1.Dadd IDN = 15 Value = 7

   Rem  Configure the AT for VFb, PFb, and PExt (IDNs 40, 51, 53)
   WriteIDNValue Drive = A1.Dadd IDN = 16 Value = 40,51,53

   Rem    Configure the MDT for VCmd, PCmd (IDNs 36, 47)
   WriteIDNValue Drive = A1.Dadd IDN = 24 Value = 36,47

   Rem   Set Sercos.Phase to 4
   Sercos.Phase = 4
 End if

End Sub
```

# APPENDIX B

## *NON-HOMOGENOUS GROUPS*

Non-homogenous groups consist of different axes, both rotary and linear. A feature was added to support the definition of velocities for non-homogenous groups. In the setup, rotary and linear axes are declared. The system determines if it is a linear- or rotary-dominant group. For example, in SCARA robot, there are two rotary axes for the first and second joints and another rotary axis for the last joint (roll). There is only one linear axis (the Z axis). Such a system is rotary-dominant.

Another example is the OCP system, which is an XY Gantry system with the last two axes taken from the SCARA. Here, there are two linear (X & Y) axes and another linear (Z axis) with just one rotary axis (roll). In this case, the system is linearly-dominant.

Axis setup defines the group behavior. Each axis must be declared whether it is linear or rotary by setting **POSITIONROLLOVERENABLE** to either zero(0) for linear or one (1) for rotary.

## Kinematics

In non-homogenous groups, commanded velocity is always given according to the group dominancy type. If the group is linearly-dominant, the velocity (**VCRUISE**, **VFINAL**) is in linear units (mm/sec). The other non-linear axes are checked in the preparation phase of the movement. The velocity in these axes cannot exceed the maximum value. If it does, the overall (group) velocity is reduced.

There is only one exception to this rule. This is when a group movement is issued with only one axis in motion and all the others are stopped. Then, the given value of the velocity is taken directly in the units for that axis. For example, having movement on the third axis of the SCARA robot, the velocity value is in mm/sec.

The same is true for both acceleration and jerk values for group-interpolated motion.

## *COORDINATE SYSTEMS*

Having different world and joint coordinates make the robot kinematics unique. The world coordinates are normally perceived as working coordinates of an application. Usually, there is some form of Cartesian coordinate system originating in the robot base.

Another set of coordinates is added for the orientation of the end-effector (gripper, etc.). The orientation of a body in space is normally described by three angles. Depending on the number of degrees of freedom (motors used to actuate the orientation joints), they could be unequal. The world space coordinates are given by its position part (X, XY or XYZ tupple), in most cases. In general, it could be anything from sphere coordinates to cylindrical system. The orientation component can use many forms for describing orientation. The most used coordinate system for orientation angles are Euler angels: yaw, pitch and roll. The Euler angles can have two or more variations, according to the order of the accepted rotations (XYX, XZX etc.).

The joint space of a robot is a relatively simple concept. A joint coordinate is a number uniquely describing the position of a robot segment relative to the previous robot segment. If the joint connecting the segment is rotary, the joint coordinates are in degrees of the rotation angle. If it is linear, the joint coordinates are the linear displacement (millimeters).

The concept of joint coordinates is an *n*-tupple number. Each number represents the coordinate of the specific joint. The world coordinates are supported by different built-in types (XY, XYZ, XYZR) that describe each of the different world space representations. Theoretically, there can be different world space descriptions for the same (or same dimensional) world coordinates. Each robot has its default world-space type. Different robots could have same world space types, as long as they have same number of degrees of freedom (**NDOF**). For example, xy-table world space has XY world space descriptors. A SCARA robot having a Z axis for vertical motion and a roll axis for orientation uses world frame coordinates consisting of: X, Y, Z and roll or an XYZR descriptor.

The different coordinate systems are implemented as variables of different data types. Generally, a point in a coordinate system is defined as a point data type. Depending on if it is a joint space or world space, the sub-type of the point differs. There is a point data type with two subtypes: JOINT and LOCATION. Both have a world space descriptor that additionally differentiates between the same subtypes. A world space coordinate is stored in a variable and is defined as:

```
DIM A AS LOCATION OF XYZR
```

The joint space is:

```
DIM A AS JOINT OF XYZR
```

The new point data type with its two subtypes (LOCATION and JOINT) and the world-space descriptor (XY, XYZ, XYZR, etc.) covers all the possible robot space variations. These variables are independent of the robot and can be used without having a robot defined in the system. When they are used in a connection with the robot or a group, the world-subtypes must be equal.

Internally, points are stored as 10-dimensional vectors with a flag describing the subtype and a size defining the dimension size of a point. The point can be manipulated as a numeric data type using all the arithmetic operations available (+-*/). The operations are defined as coordinate-by-coordinate operations. Only points of the same subtype and same world-space can be combined.

Special syntax is used for point constants: a list of values separated by commands inside squared brackets ({}) denote points. If it is preceded by pound sign (#), it is a location:

```
DIM C AS LOCATION OF XYZR
C = #{0,0,0,0}
DIM I AS JOINT OF XYZR
I = {0,0,0,0}
```

## *USER UNITS*

The MC gives you the freedom to change and choose different units. Robot models have less flexibility. Due to a tight relation between joint and world-space, you must define joint position units in millimeters or degrees and all time units in seconds so both world space and joint space are compatible units. The units of world space are automatically set to millimeters-degrees-seconds.

Each axis has four scaling factors: **PFAC**,**VFAC**, **AFAC**, and **JFAC**. These translate encoder/resolver counts into user position units (inches, millimeters or revolutions). They also specify what time units are used in position derivations (**VEL**, **ACC**, **JERK**). As the position factor defines the ratio between counts and user position units, the rest of the factors define the ratio between milliseconds and user time units. Typically you have:

```
<axis>.pfac = <number of counts per user position units>
<axis>.vfac = pfac/1000 (if in seconds)
<axis>.afac = vfac/1000 (if in seconds)
<axis>.jfac = afac/1000 (if in seconds)
```

Once all axes are set up, groups contain these axes. You must have the ***same user units*** on all axes. Set the group scaling factors. **There is no position factor for the group!** Instead, **PFAC** for each axis is used.

```
MOVE g1 {1,2,3} vcruise = 10
```

Target coordinates are given to each axis and translated into encoder/resolver counts using **PFAC** for each axis. **VCRUISE** is not related to any axis. Group factors, **VFAC**, **AFAC**, and **JFAC** are used. The group velocity is expressed in position user units per seconds.

To have system units properly working, set the unit factors (**PFAC**, **VFAC**, **AFAC** and **JFAC)** of each axis to millimeters-degrees-seconds and set group units to seconds:

```
<group>.vfac = 1/1000
<group>.afac = vfac/1000
<group>.jfac = afac/1000
```

## *COUPLING*

In many robotics applications, mechanical construction introduced mechanical coupling between the axes. In such cases, two or more motors move only one joint. So activating only one motor causes motion in two axes (joints). There are many examples of such mechanical setups. Usually when the motors are dislocated for the joints, they are actuating. When the motor-actuating second joint is placed before the first joint, the second joint is moved by the motions of the first and second motors. Typical examples of such applications are displaced motors of a robotic arm (PUMA) moving the robot arm via chains or belts. Differential gearing on the last robot segment displaces the last motor (roll) from the end effector and both the fifth and sixth motors introduce motions on the sixth and with joint (Staubli RX series).

Staubli RX – pair axes a5 ,a6:

```
j6.pcmd = a6.pcmd – a5.pcmd
```

BOSCH – SCARA – pair axes a3,a4:

```
j3.pcmd = a3.pcmd + 18.3*a4.pcmd.
```

PUMA – a2,a3 and a4:

```
j4.pcmd=a2.pcmd+a3.pcmd+a4.pcmd.
```

Scaling and Rotation:

```
    j1 = 707.1*a1.pcmd + 707.1*a2.pcmd
     j2 = 707.1*a1.pcmd + 707.1*a2.pcmd
```

Orthogonal Correction:

```
   j1 =10000*a1.pcmd
   j2 = 2.91*a1.pcmd + 10000*a2.pcmd
```

Group joints are standard in the MC's firmware, covering coupling examples given above.

## *MOTION ELEMENTS*

The MC is a multi-axis and multi-group system. In many aspects, groups and axes are the same. For example, both have properties like: **AMAX**, **VMAX**, **VCRUISE**. Another common aspect is the movement in both of them. **MOVE** moves both axes and groups. Although, there are motion commands that are group-only (**CIRCLE**, **MOVES**) or axis-only (**JOG**), when moving an axis separately, it is not possible to move the group containing the same axis and vice versa. The axis is active in a group. **ATTACH** and **DETACH** work the same way. You cannot attach an axis when its group is already attached. So, we can denote both axes and groups using one term, motion-element.

There are some properties that are axis- or group-only properties. For example **DRIVEADDRESS** is an axis-only property.

**MOVE** consists two different command behaviors, Move-Group and Move-Axis. Move-Group moves the whole group together. All the axes within the group start and end the motion at the same time. Move-Group stays within the kinematics limits (**VELOCITY**, **ACCELERATION**, **JERK**) of the group and axes. The group limits are taken and imposed on the group motion (aggregate, space – XY, XYZ). The axis limits are imposed only proportionally to the ratio of the axis motion to the whole group motion. If axis X is moving halfway toward axis Y, only half the limit values are used. The total time of the motion is the time required for the slowest axis to accomplish the movement. Here, the slowest axis is the one with the largest path/velocity ratio and not necessary the axis with smallest maximum velocity (because if such an axis has no movement to make, it is not the slowest axis). On the other hand, Move-Axis checks only the limits of the specified axis and is independent of the limits of any group using this axis.

Profiler preparations can additionally reduce the motion parameter. This is typical for short movements where the given cruise velocity cannot be reached due to existing limitations of **ACCELERATION** and **JERK**.

# Joints

Joints are virtual axes. Joints give the illusion of only one axis, both from the language point of view and looking at the physical movement of the coupled axis. The joint is analogous to an axis. Joints are denoted as J1, J2, etc.

Each joint actually represents one or several axes of the group, according to the given coupling matrix. If the group has no coupling matrix defined, the joint is a direct representation of the axis of the same ordinal number in the group as the joint index. To introduce another term, the "shadow-axis" of a joint is the axis with the same ordinal number in the group as the joint index. For example, if a group consists of axes A2, A3, A4, the shadow-axis of the joint J1 is A2.

Normally, all joint properties are a direct representation (have the same value) of shadow-axis properties, independent of whether the coupling matrix is defined or not. The only exceptions are properties related to joint position: **PCMD**, **PFB**, **VCMD**, **VFB**, **CCMD**, **CFB**, **PMAX**, and **PMIN**. These are unique for each joint.

Current positions and velocities are obtained according to the given coupling matrix. They are computed each time a query of these values is issued (**PCMD**, **PFB**, etc.). They are read-only and do not represent any specific internal variable. Symmetrically the same properties, when queried from a group (`?g1.PCMD`), return lists of joint-property values according to the coupling matrix. The movement's target positions (**MOVE** or **CIRCLE**) of group movements are treated analogously (coupling matrix).

On the other hand, the joint's position limits (**PMAX** and **PMIN**) are added to the joints. The position limit of the joint and the position limit of the shadow-axis are not related, but independent of whether the coupling matrix is defined or not. The joint position limits are always used when the coupling matrix is defined (`COUPLED = 1`). In this case, the shadow-axis limits are not used. This is only true for joint and group movements. Single-axis movement (e.g., `MOVE A1 100`) is limited only by the axis limits, independent of coupling.

Joints represent several axes of a group. Joint movements are actually group movements:

```
Move-Group = Move-Joint
```

Joint movement gives the illusion of moving one physical axis (`MOVE J1 100`), although several motors could be moving together. The movement is executed according to the specified joint's movement parameters. For example:

```
A1.Acc = 100
MOVE A1 50
```

Is analogous to:

```
J1.Acc = 100
MOVE J1 50
```

Move-Joint simulates the behavior of moving a single axis by moving a single joint both by the parameters and physical motion. The only difference is that several motors are moving instead of just one. When moving joints, all motor limits within each group axis are checked. The joint movement is a group movement of one joint-coordinate with the group motion parameters copied from the shadow-axis. The following translation rule is true:

```
MOVE J1 10
' MOVE {10,0,0,0} abs = J1.abs vcruise = J1.vcruise acc = J1.acc
. . .
```

The same rules of group-motion apply.

# Robot Models

A robot is defined as a group with a special model type. Currently, the following kinematics models are available:

1 – regular group: no kinematic model (default value)
2 – PUMA: 6 axis, XYZYPR (X,Y,Z, Yaw, Pitch, Roll)
4 – SCARA: 4 axis, XYZR (X,Y,Z, Roll)
6 – DELTA: 4 axis, XYZR (X,Y,Z, Roll)

```
COMMON SHARED scara AS GROUP AxNm = a1 AxNm = a2 AxNm = a3 AxNm = a4 Model = 4
```

This automatically defines a robot with a SCARA kinematics model. Contrary to the regular groups that need only the axis to be enabled, you must set the robot parameters and configure them. The robot parameters needing to be set differs from robot to robot. For the SCARA robot, these are:

Set all axes with all standard kinematics parameters.
Define the roll-over properties.
Set all user-units factors for the entire group: **VFAC**, **AFAC**, **JFAC**
Set segment lengths and joints orientations for each robotic link.
Set kinematics of regular groups (**VELOCITY**, **ACCELERATION**, **JERK**, etc.)
Set kinematics of Cartesian motions (**VTRAN**,**VROT**, etc.)
Issue **CONFIGGROUP**.

Only after setting all these and successfully executing **CONFIGGROUP** can the robot be moved. Otherwise, an error is returned, stating that the group is not configured. The important difference between the regular groups (those with default model value 1) and the robot, is the necessity of executing **CONFIGGROUP**. In robot groups, no motion is allowed before the robot is configured.

Model value automatically defines the default robot world space descriptor. For model 4 (SCARA), the XYZR (x-y-z-roll) world-space is selected. With no robot model defined (model=1), the world-space descriptor is chosen according to the group size:

2-axes: XY
3-axes: XYZ
4-axes: XYZR

In these cases, world-space is identical to joint-space. Each coordinate of the joint-space vector is directly translated into a coordinate of world-space.

In groups with the robot model defined (model > 1), the translation between joint- and world-space is much more complex. Translating joint coordinates into world coordinates is accomplished with **TOCART** taking both the robot and given joint coordinates (position variable) as input arguments. The output of this is world coordinates corresponding to the given joint position.

The other direction is more complex. It is accomplished with **TOJOINT** receiving three input arguments: the robot, the given Cartesian point, and the configuration flag.

The configuration flag is used to select between the two translation functions. In most robot models, there is more than one joint coordinate description for the same world-space position. The different robot configurations can produce the same world-space positions with different joint-space values. The robot configurations differ from robot to robot. In SCARA, there are only two configurations: lefty (second joint coordinate is negative) and righty (second joint coordinate is positive).

# Interpolation

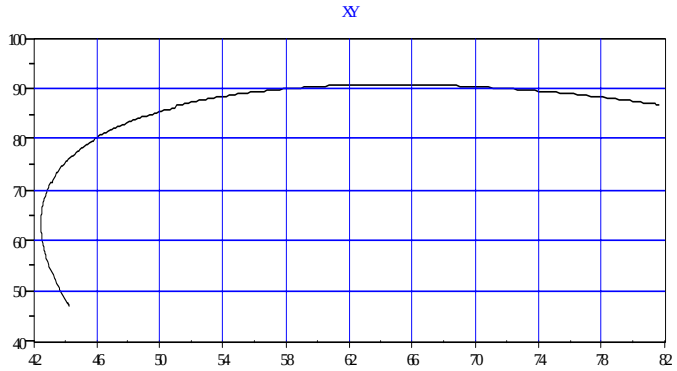
For groups with robot models (model > 1), there is an additional moving type: **MOVES**. The movement makes a straight path from the starting position to the given target position in the world space. The straightness is tightly-related to the working space. The same path can be straight in the Cartesian working space (XYZ) but curvy in joint space, and vice versa. **MOVES** is used solely for groups from type robot and cannot be used in regular groups.

Straight line motion refers to motions that are straight lines in world-space. In groups with no robot model (model=1), all motions produced with **MOVES** are straight-line motions in joint-space. In regular linearly-dominant groups, most of their axes are linear. As the world-space and joint-space are identical, **MOVES** executes a straight line in both joint- and world-space. The target position can be given both in world- and joint-space coordinates. The movement differs from joint interpolation (**MOVE**). Instead of joint-space kinematics parameters used in regular groups, the world-space parameters for velocity, acceleration and jerk are used. These parameters are available in two forms:, one for the translational part of the motion and one for the rotational part (orientation).

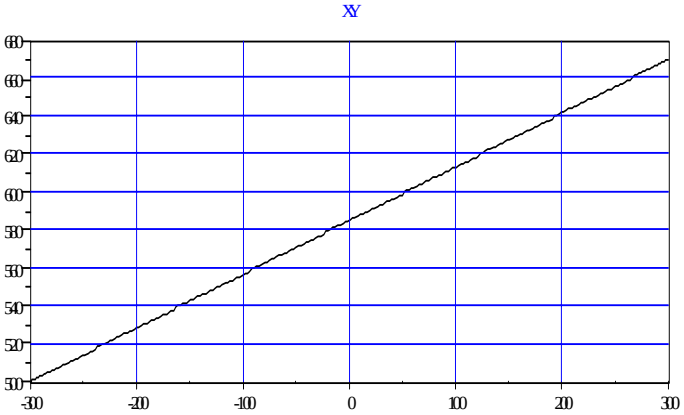
When world-space has both position and orientation components, the straight-line motion includes interpolation on both position and orientation. Because these two are intrinsically different objects, two different parameters are needed: one for the translation (pure position interpolation): **VELOCITYTRANS**, **ACCELERATIONTRANS**, **JERKTRANS**; and one for the orientation (rotation interpolation): **VELOCITYROT**, **ACCLERATIONROT**, **JERKROT**. These parameter are used only for Cartesian interpolation (**MOVES** and **CIRCLE**). These parameters have no influence on joint interpolated motion (**MOVE**) because it is not a world-space interpolation.

The two sets of parameters define the straight line motion. Which parameter is dominant for the motion depends on the movement. If most of the movement is in translational position change and the orientation change is much smaller, the translational part is taken. Conversely, the rotational parameters define the movement.

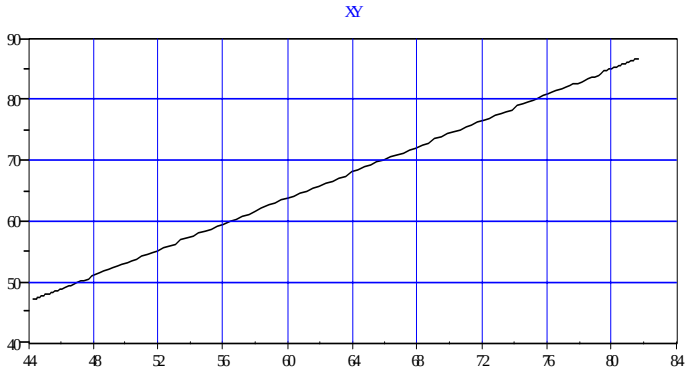
The following graph shows **MOVES** as seen in joint-space.



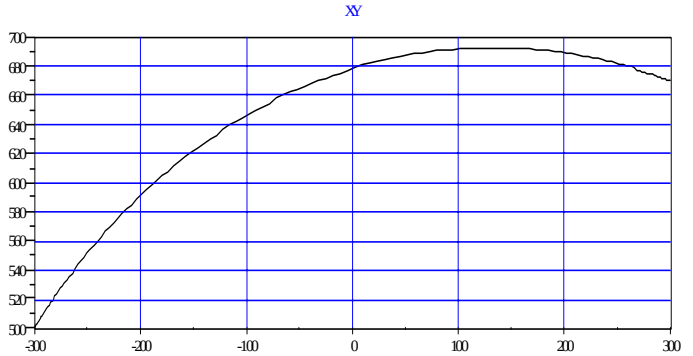

The following graph shows **MOVES** as seen in world-space.



The following graph shows **MOVE** as seen in joint-space.




The following graph shows **MOVES** as seen in world-space.




There are four commands to make point-to-point movement in robot groups. There are two commands to interpolate the motion (**MOVE** and **MOVES**) and two commands to define the target point (**JOINT** and **LOCATION**). The total combination of these gives four options.

| | **LOCATION target** | **JOINT target** |
|---|---|---|
| **Cartesian-Interpolation** | MOVES #{400,400,0,0} | MOVES {45,45,0,0} |
| **Joint-Interpolation** | MOVE #{400,400,0,0} | MOVE {45,45,0,0} |

## *CONTOURING*

Contouring connects different motions by defining the final velocity for each movement. This way, the motion does not stop at the segment. It is continued with the next motion. This method is applicable in one-axis applications, but has several drawbacks that become more prominent in multi-axis applications:

The angle between the movements' direction must be zero. Otherwise, a jump in velocity occurs.

It is not possible to connect arc with linear segments without a jump in acceleration.

You must calculate the achievable final velocities on each segment to stop after the last segment.

# Cartesian Profile

For kinematics parameters of straight-line (**MOVES**) motion (velocity, acceleration, deceleration), different joint parameters are obtained, depending on where in the robot working space the starting line is positioned. The further from the origin, the smaller joint velocities are obtained. Contrary to that, the closer to the origin the point is, the smaller the inertia and higher the acceleration.

To optimally exploit robot capabilities, execute motions nearer to the origin with smaller Cartesian velocities to avoid exceeding the maximum joint velocity. Movements near the limits of the working space are farther from the robot origin and have higher robot inertia. Lower acceleration must be applied to avoid exceeding the maximum joint values.

You must know at least two values (threshold of arm radius where reduction begins and end point of reduction at straight arm) for all joints. Limits must be considered for the joint. The whole Cartesian path and all Cartesian interpolation types (straight, circular, via) must be considered.

Since joint velocity, acceleration, or jerk values cannot be exceeded, take the minimum of the commanded Cartesian values (**VTRAN**, **ATRAN**, **DTRAN**, **JERKTRAN**, **VROT**, **AROT**, **DROT** and **JERKROT**) and account for the values in joint-space (**VCRUISE**, **ACC**, **DEC**, **JERK**).

Cruise velocity cannot vary. Use a constant cruise value. Leave the principal profile acceleration and deceleration phase as it is. Acceleration and deceleration cannot vary during the constant **ACC/DEC** phase. Use the highest acceleration in Cartesian space possible that allows constant **ACC/DEC** phases without exceeding joint values. Acceleration and deceleration may differ, and must be considered separately (mostly in long distance moves where conditions vary widely for start and destination position). The same restrictions apply to **JERK**.

Acceleration, deceleration and jerk reduction must be calculated and anticipated, depending on payload and arm radius. The parameters for a simple scheme are:

threshold values for payload and arm radius where reduction begins

overall 100% reduction value per robot group

per-joint 100% reduction values

Since payload is static (varying only with tool change), this is easily accomplished in the application. Use the following two mechanisms for Cartesian profiles:

**C**artesian **V**elocity **A**daptation (CVA) is according to the direction and position of the movement in the working space. For each path, a point closest to the origin (X=0,Y=0) is found and the Cartesian velocity of the whole motion is reduced according to the values of joint velocities at that point.

**C**artesian **A**cceleration/**D**eceleration/**J**erk **A**daptation (CADJA), where values of jerk, acceleration and deceleration are reduced according to the relative distance from the point of origin in reference to the maximum radius of the robot. The reduction (**INERTIATHRESHOLD**) is active only from a certain distance (radius). The maximum reduction value is defined at the final point (the fully stretched arm). For all points between these two values (INERTIATHRESHOLD = 100%), the reduction value is linearly interpolated. Acceleration is reduced according to the initial point of the movement. Deceleration is reduced according to the final point of the movement. The jerk value is reduced according to the average of the two reduction values (for initial and final point).

Limitations are:

Reduction factors are not allowed for individual axes – only for groups.

The CVA works only for **MOVES** & **PASS-THRUGH** .

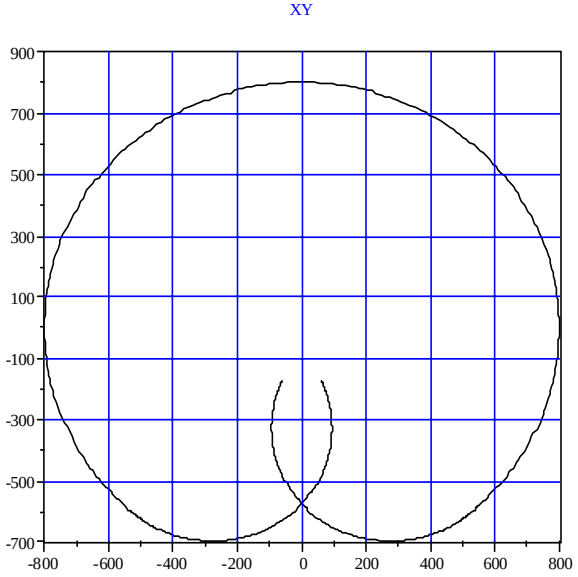
The CADJA works only for **MOVES** & **PASS-THRUGH**. In **PASS-THROUGH**, only the first and last point are treated.

Both methods work only for SCARA kinematics.

# Working Envelope

Robot working space is determined by the position limits of each joint. The working space (all reachable positions) is limited by the working envelope of the robot. From the outer side, it is limited by the arc made rotating the first joint with a fully stretched arm (`j2.pcmd=0`). This is the maximum radius of reachable points (**RMAX**). It is an internal value computed each time the robot is configured (**CONFIGGROUP**). Points given outside this radius are rejected with the error message, *Point Too Far*.

The other limit is the small circle with the radius (**RMIN**). This circle is obtained by rotating the first joint with a maximally-folded second joint (not more then 180 degrees). This radius is additionally enlarged by the robot's base size and the attached end-effector mechanism to prevent the robot from colliding with itself. The minimal working envelope radius (**RMIN**) is available to the user (contrary to **RMAX**). Danaher Motion recommends setting this value for each application, according to the physical setup of the robot. If a point is given inside **RMIN**, an error message is returned. For example:



```
Program
   attach
   move {j1.pmin,j2.pmin,0,0}
   while ismoving
      sleep 1
   End While
   Print "Plotting the workspace"
   move {j1.pmin,0,0,0}
   move {j1.pmax,0,0,0}
   move {j1.pmax,j2.pmax,0,0}
   detach
End Program
```

# Robot Configurations

In robot models, there are several joint points representing the same robot end-effector position for each location point. To uniquely select between different joint coordinates, there are configuration flags.

For SCARA kinematics, there is only one configuration flag available: the arm flag. It indicates either lefty (1) (`j2.pcmd > 0`) or righty (2) (`j2.pcmd > 0`). A value of 1 (lefty) means that the joint coordinates having a positive second joint are taken to represent the given location. The current configuration of the robot is returned by **ARMFBK**. The value can be 1 or 2, depending on the position feedback value of the second joint. **ARMFBK** determines the movement's target position. It can be automatic (0), lefty(1) or righty(2). When a movement with a Cartesian target position is given, the joint coordinates of the target position are selected according to **ARMFBK**. If it is zero (automatic), the current robot configuration is used (**ARMFBK**).

PUMA models traditionally use three configuration flags (arm,elbow and wrist – **ACMD**, **ECMD**, **WCMD**).

Another use of configuration flags is in the conversion function, **TOJOINT**. This function translates the given location point into joint point. It receives two arguments. The first is the location variable and the second is the configuration flag.

# *POINTS*

A point is a new data type used for storing a list of doubles in a variable. The main advantage of points is that you can directly call the whole vector without having to access every element of it. There are two subtypes of the point data type: LOCATION and JOINT. Using both point types in the same expression results in a type mismatch translation error. Type casting between point types requires the usage of special system functions **TOJOINT** and **TOCART**.

# Declaration

A point must be declared before you can use it. The declaration of points is defined according to the type of robot (XYZR, XYZ, etc.). The MC enables the declaration of point variables, which can be scalar or arrays. Point arrays may have up to 10 dimensions. Point variables are designed to hold a list of 2 to 10 double-type coordinates. A point variable is declared in relation to a robot type. Therefore, declaring a point variable must include the name of a valid robot type. The dimension of the point is the same as the dimension of the type of robot. The syntax is:

```
(COMMON SHARED|DIM {SHARED}) <variable_name> AS JOINT OF <robot_type>
(COMMON SHARED|DIM {SHARED}) <variable_name> AS LOCATION OF <robot_type>
```

<*robot_type*> can be:

XYZR – Three cartesian axes + roll
XY – two axes XY table
XYZ – three axes XYZ system

For example:

```
Common Shared JointXYZ As Joint Of XYZ
Common Shared JointXYZR As Joint Of XYZR
Common Shared LocXYZ As Location Of XYZ
JointXYZ = LocXYZ                              → Error – type mismatch
```

# Variables

Two new variable types are defined for point variables in the translator: LOCATION and JOINT. For the translator, a point variable differs from any other type of variable (Long, Double, etc.) only by its type. A point variable is inserted to the expression tree as a variable leaf. The variable leaf retains data related to the point variable, including its context (system, program or local), its offset (taken from the symbol table), and its type (LOCATION or JOINT, also taken from the symbol table).

## *DECLARATION*

When declaring a point variable, data corresponding to this variable is entered into the correct symbol table (depending on whether it is a system variable, a program variable, or a local variable). A point variable is defined in the symbol table by its name, its type (LOCATION or JOINT) and an offset of the data segment. Each offset of the data segment corresponds to the address of the point variable. The robot type string is sent to a special function that gives the robot-type value and the number of coordinates.

# Constant Points

Constant points are location or joint vectors with an undefined robot type. The constant point's subtype is defined by the shape of its brackets (#{ } for location, { } for joint). In the translator, each of these two vector types is considered an expression. Each expression-related command and manipulation (such as print commands, mathematical operators, etc.) can be applied for the joint and location vectors.

While executing a vector expression, each vector element is pushed separately into the stack as a double type expression (by executing the left side of the JOINT or LOCATION node). Long type vector elements are converted to double type expressions (in the coordinate node) after the execution of the expression at the right side of each coordinate node).

Execution of the right side of the JOINT or LOCATION node results in pushing the number of the vector elements into the stack (as a long type constant). The point's subtype (determined by the shape of the vector's brackets) is pushed to the stack (as a long type constant) by executing the third side of the JOINT/LOCATION node.

# Vectors

Vectors are actually constant points, at which a list of 2 to 10 coordinates is written between curly brackets {}. In location vectors, the curly brackets are preceded by the # sign. A coordinate can be any type of expression (constant, variable, mathematical expression, function call, etc.) of long or double type. Long type coordinates are automatically converted to doubles. Unlike point variables, vectors are not related to robot types, and characterized merely by their size (number of coordinates).

```
{<expression_1>, …, <expression_n>}                ' Joint vector
#{<expression_1>, …, <expression_n>}               ' Location vector
```

Point variables and vectors appearing in the same expression must match in type (joint or location) and size.

```
Common Shared JointXYZ As Joint Of XYZ
JointXYZ = #{1.5, 2.8, 3}                    → Error – type mismatch
PRINT {20, 6.7, 23} + #{8.43, 2, 0}          → Error – type mismatch

JointXYZ = {2, 6}                            → Error – size mismatch
PRINT {20, 6.7, 23} - {5, 4, 22.9, 31}       → Error – size mismatch
```

# Properties

Some group properties are points. There are several read-only properties of joint-(**VELOCITYFEEDBACK**, **POSITIONCOMMAND**, **DEST_JOINT**), and location-type (**DEST**, **HERE**, **SETPOIN**T). There are also some location-type read-write properties, like **TOOL**, **BASE**, etc.

As in point variables, the robot type also characterizes point properties.Point properties appearing in expressions in a mixture with other points must match in type and robot type to point properties and variables, or in type and size to vectors.

```
Common Shared SCARA As Group Axnm = A1 Axnm = A2 Axnm = A3 Axnm = A4 Model = 4
Common Shared PointXYZR As Joint Of XYZR
Dim Shared PointXYZ As Joint Of XYZ

PointXYZR = SCARA.SetPoint
                → Error – type mismatch (Setpoint returns a location)
PointXYZ = SCARA.VelocityFeedback          → Error – robot type mismatch
SCARA.Base = #{56.5, -104.7, -90.5}        → Error – size mismatch
```

# Point Dimension

You do not define the size of the vector data type at the declaration of it, but you define the type of robot related to the point. This gives the point a dimension.

# Point Assignment

Point assignment uses the following syntax:

```
<variable_name> = (<expr1>,<expr2>,<expr3>,…,<exprn>)
() = #{} for location
() = {} for joint
```

The expression can be a constant or other vector variable of the same robot type, but the size of the right side of the expression must equal the left side. If you try to assign a point with a point of other robot type, an error is received.

# Single Coordinate Point Assignment

The syntax for assigning a single coordinate of a point is:

```
<variable_name>{<expression>} = <expression>
```

# Point Query

The syntax for querying a point is:

```
?<variable_name>
```

# Single Coordinate Point Query

The syntax for querying a single coordinate of a point is:

```
?<variable_name>{<expression>}
```

**Example1:**

```
Common shared P1 as location of XYZR   'declaration of a vector (global)
P1 = #{1,2,3,4}                 'assign the whole vector
P1 = #{1,2}                     'return a translation error
?P1                             'read the whole vector
?P1{2}                          'read the second coordinate of the vector
```

**Example2:**

```
Common shared P2[5] as location  XYZ   'declare a global array of vectors
P2[1] = #{1,2,3}                  'assign the whole vector
P2[2] = #{5,2}                    'return a translation error
?P2[1]                             'read the whole vector
?P2[2]{2}                          'read the second value of the vector
```

# Operators

Like any other data type, you can use operators for points. The system only defines the equal operator (plus, minus, multiplication and division). Operations between points can only be performed between points of the same type and robot type (or size). All operations can also be performed between points and long or double type expressions.

## *ASSIGNMENT*

Point variables and read-write properties can be assigned by point variables, vectors and point properties compatible in type and size (or robot type).

## QUERY

Point variables, properties and vectors return a value, so they can all be queried through printing or assignment into a compatible point variable or read-write property.

```
Common Shared SCARA As Group Axnm = A1 Axnm = A2 Axnm = A3 Axnm = A4 Model = 4
Common Shared PointVar1 As Joint Of XYZR
Dim Shared PointVar2 As Joint Of XYZ
Dim PointArr[4] As Joint Of XYZR

PointVar1 = PointArr[2]
PointVar2 = {23.4, 60, 42}
PointArr[1] = SCARA.PositionCommand
SCARA.Tool =  #{10, 20,30, 0}
PRINT PointVar2, SCARA.Tool, {100.3, 20}
   →   {23.4 , 60 , 42}   #{10 , 20 , 30 , 0}   {100.3 , 20}
```

## PLUS

The point plus operator is used like any other data type. It adds each element of the first point with each element of the second point. The result is a point.

```
?(1,2)+(2,4)              → (3,6)
```

## MINUS

The point minus operator is used like any other data type. It subtracts each element of the first point with each element of the second point. The result is a point.

```
?(1,2)-(2,4)              → (-1,-2)
```

## MULTIPLICATION

The point multiplication operator is used like any other data type. It multiplies each element of the first point with a number. You can only multiply a point with a single number (one-dimensional number). The result is a point.

```
?2*(2,4)              → (4,8)
```

## DIVISION

The point division operator is used like any other data type. It divides each element of the first point with a number. You can only divide a point with a single number (one-dimensional number). The result is a point.

```
?(2,4)/2              → (1,2)
?2/(2,4)              → (1,0,5)
```

## COMPOUND (:)

Is an operator specific for points, which should be operated between two locations of the same size (or robot type).

```
? #{-56.5 , -104.7 , 89.5} : #{ 0.0, -104.7 , -0.5}
     → #{-56.5 , -209.4 , 89}
```

## LIMITATIONS

MOD (modulus), ^ (power), logic and bitwise operators cannot be used for points.
Points cannot be used as condition in flow control statements and event definitions.
Points cannot be recorded.

# Points in Functions

Point variables can be passed to functions and subroutines both by reference and by value. An entire array of points can also be passed (by reference) to a function or subroutine. On the other hand, point properties and vectors can only be passed by value. Points can also serve as return values of functions. However, the point type and size (or robot type) of an argument or a return value must match function declaration.

## *PASSING BY VALUE AND REFERENCE*

A point can be passed by value and by reference like any other data type. For example:

```
Program
Dim p1 as joint of XYZR
Call Sub1(P1)
?p1
End program
Sub Sub1(X as joint of XYZR)
X = {1,2,3,4}
End Sub
```

This program assigns P1 to {1, 2, 3, 4} and prints {1, 2, 3, 4} as output. When a point is passed by reference or value, you can only pass a point of the same robot type to *X*. Otherwise, you receive a translation error.

**Example1:**

```
Program
Dim p1 as location of XYZR
Call Sub1(P1                                    → TRANSLATION ERROR !!!
?p1
End program
Sub Sub1(X as location of XYZ)
X = #{1,2,3,4}
End Sub
```

**Example2:**

```
Program
Dim p1 as joint of XYZR
Call Sub1(P1)                                   → TRANSLATION ERROR !!!
?p1
End program
Sub Sub1 (ByVal X as joint of XYZ)
X = {1,2,3,4}
End Sub
```

## *RETURNING POINT FROM FUNCTION*

A function can return a point variable like any other data type. For example:

```
Dim A as location of XYZR
Program
Move G1 MyFunc(1) vcruise=299
        → this will be OK if G1 is of XYZR robot type.
End Program

Function MyFunc(i1 as long) as location of XYZR
MyFunc = #{1,2,3,4}
End Function
```

# Motion Commands

Some motion commands use point nodal parameters when applied on groups. For example: **CIRCLE**, **MOVE** and **MOVES**.

Some nodal parameters can be assigned with both point types (LOCATION and JOINT), like **CIRCLECENTER**, **CIRCLEPOINT** and the target positions of **MOVE** and **MOVES**. Other nodal parameters can accept only location points, like TOOL, BASE, etc. Assign all nodal parameters with points compatible in robot type or size to the group.

```
Common Shared SCARA As Group Axnm = A1 Axnm = A2 Axnm = A3 Axnm = A4 Model = 4
Common Shared JointXYZ As Joint Of XYZ

MOVE SCARA JointXYZ                              → Error – robot type mismatch
CIRCLE SCARA Angle = 180 CircleCenter = #{350, 500, -80}
    → Error – size mismatch
MOVES SCARA #{-22, 503.8, -8.3, 149} Base = SCARA.Dest_Joint
    → Error – type mismatch
```

# Analog Behavior

Because a point is a new data type, it has all the functionality of the other data types like saving the point to a file, watching points, deleting points.

# Point As An Expression

With the new point data type, points properties can be stored as a point or can be compared as a point. For example:

```
Dim A as Joint of XYZR
A = {1,2,3,4}
A = G1.Pcmd
A = G2.Pcmd          → returns a translation error if G2 is not of XYZR type.
```

## *FUNCTIONS*

The following functions are defined:

```
<XYZR location> = TOCart (<group>, <XYZR JOINT>)
<XYZR Joint> = ToJoint (<group>, <XYZR LOCATION>, <arm-flag:integer>)
<double> distl(<location>,<location>)    Distance between points (length).
<double> distr(<location>,<location>)       Distance between points (angle).
```

# *SAVE/LOAD*

Each global variable can be stored in a file. The variable and its value are stored in same syntax as it is defined in the program. You can select a data-type to be stored. If a data-type is selected, the SAVE command stores only global variables of that data type. If file with same name already exist its extension is renamed to *.bak and new file is created. Loading the file overwrites the values of the saved global variables. You can save all the points related to some type of robot. For example:

```
SAVE file = "Myfile.prg" type = location robottype = xyzr
SAVE file = "Myfile.prg" type = all
LOAD Myfile.prg
```

# Group Point Properties

Following modal-only and read-only group properties of the point data-type exist:

| | | |
|---|---|---|
| DEST | DEST_JOINT | Target point of the movement. |
| START | START_JOINT | Starting point of the movement. |
| HERE | POSITIONFEEDBACK or PFB | Actual point of the robot (feedback). |
| SETPOINT | POSITIONCOMMAND or PCMD | Actual point of the robot (group). |

## *SINGLE ELEMENT ASSIGNMENT*

Syntax for assigning a single element of a structure:

*<variable_name>-><structure element> =<expression>*

# In Functions

Structures can be passed to functions and subroutines both by reference and by value. Also, an entire array of structures can be passed (by reference) to a function or subroutine. Structures can also serve as returned values of functions. However, the structure type of an argument or a returned value must match function declaration.

```
Dim Shared ARG1 As Type_1
Dim Shared ARR_ARG[4][5] As Type_1
Dim Shared ARG2 As Type_2

FUNCTION MyStructFunc(BYVAL Par1 AS Type_1, Par2 AS Type_2) AS Type_1
…
MyStructFunc = Par2                    ->   Error - structure type mismatch
MyStructFunc = Par1                    ->   Structure types match
END FUNCTION

SUB MySub(Arr_Par[*][*] AS Type_1)
…
END SUB

ARR_ARG[2][3] = MyStructFunc(ARG1, ARG2)   ->   Structure types match
ARR_ARG[2][3] = MyStructFunc(ARG1, ARG1)   ->   Error - structure type
mismatch
CALL MySub(ARR_ARG)
```

or

```
Common Shared ST As STRUCT
Common Shared LongVar As Long

FUNCTION MyFunc(BYVAL Par1 AS LONG, Par2 AS LONG) AS STRUCT
…
MyFunc->LongElm2 = 10
MyFunc->JointElm = {12.3, 5, 43.29}
END FUNCTION

FUNCTION NoParamFunc AS STRUCT
  …
END FUNCTION

ST = MyFunc(ST->LongElm1, LongVar)
ST = MyFunc(ST->LongElm1, ST->LongElm2)
                        ->   Error – structure element passed by reference

? MyFunc(ST->LongElm1, LongVar)->LongElm2
                         ->   Error – addressed through function call
LongVar = NoParamFunc->LongElm2
                        ->   Error – addressed through function call
```

A function can return a structure variable like any other data type. For example:

```
Dim Shared A as X
Program
Dim S1 as X
   S1 = MyFunc(1)
End Program

Function MyFunc(i1 as long) as X
   MyFunc->Type = 1
End Function
```

# Query

You cannot query a whole structure or you receive a translation error.

```
?<variable_name> - gives a translation error
```

## *SINGLE ELEMENT QUERY*

Syntax for querying a single element of a structure:

```
?<variable_name>-><structure element>
```

For example:

```
Common shared s1 as X           'declaration of a default structure
S1->.Type =1
S1->Length =2
?S1                             'Translation Error
?S1->Type
```

# Operators

You cannot use operators with structures. It is possible to use operators for the single elements of the structure.

# Pass Structure

A structure can be passed by value and reference like any other data type.

Example1

```
Program
Dim s1 as X
Call Sub1(S1)
   ?S1->Type
End program
Sub Sub1(X as X)
   X ->Type= 1
End Sub
```

Example2

```
Program
Dim s1 as X
Call Sub1(S1)
   ?S1->Type
End program
Sub Sub1(X as X)
   X ->Type= 1
End Sub
```

# Limitations

A whole structure cannot be printed.

Mathematical, logic and bitwise operators cannot be used for whole structures.

A whole structure cannot be used as a condition in flow control statements and event definitions.

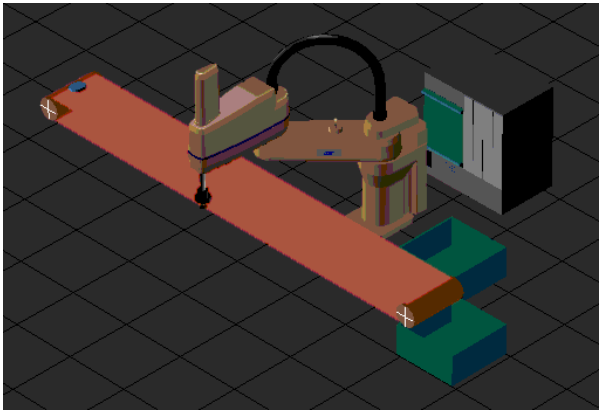
A whole structure cannot be recorded.

For example:

```
Common Shared VAR1 As STRUCT
Common Shared VAR2 As STRUCT
PRINT VAR1                                  ->   Error
VAR2 = VAR1 * 100                           ->   Error
IF VAR2 > VAR1 THEN                         ->   Error
WHILE VAR2                                  ->   Error
```

## *CONVEYER TRACKING*

A moving frame's variable data-type is defined by the user as a separate unit (object) and can be linked to one or more robots. The data type has its own properties and definitions. The moving frame represents a coordinate system relative to the robot's world coordinate system. Set a limited area (in further text: window) on which the robot tracks. This area is the operation region or the working frame. Before the entrance to this region, there is a sensor that marks objects entering the region. The tracking starts when a trigger occurs and you enable the procedure. During tracking, you can move the robot both in absolute and relative movements. The absolute movements shift by the tracking process in the same direction and distance as the moving object. The relative movements are commanded in respect to the current robot position. The relative movements are superposed to the movements caused by the tracking of the moving frame. The tracking lasts until the object exits the working frame,. Then, the robot starts tracking the next item on the frame.



You can classify objects in the working frame and only track certain types of objects. Each moving frame can have several master sources (external sources of moving position). The number of sources is the number of degrees of freedom of the moving frame. You can use several robots to track the same operating region, but the region must be presented as a different moving frame object. Robots must be of the same type as the world frame of the object in the tracking process (dimension and robot type). The process is completed in one of two ways: disable tracking or send no new triggers.

There are two different concepts of NDOF's: one for the robot which means the number of motors driving it and one for the moving frame, which means the number of external position sources assigned to it. These two numbers can be different. Only the object type (dimension and type) of the moving frame must match the robot's world frame.

# Window Declaration

Limit the region by one or more pairs of points, depending on the number of degrees of freedom of the element to be tracked. For every degree of freedom a pair of points defines its region limits. The boundaries are the **upstream** (lower limit) and the **downstream** (the upper limit). The lower and upper limits are referred to the trigger point. The lower limit is closer to the trigger point than the upper limit. Moving direction is from upper to lower. While the object is inside the working frame, every change of conveyor movement direction is tracked. The only limitation is that region limits must be relative to the trigger position so the upstream is closest to the trigger. An object might enter from the downstream limit. To correct this error, add a direction flag to the trigger to indicate if the region limits are inverse.

The moving frame is divided into components. For example, an XY table tracks as two axes: X and Y. The coordinate translation is done automatically when using the robot coordinate system as a base for the position calculation. The moving frame coordinate system is the same as the tracking robot. The master source is independent in the moving frame type. The group master is used as an input position to follow.

# Tracking

Tracking starts when the tracking process flag is enabled, which assumes that the object has passed the sensor. Tracking actually starts when the relevant object enters the operation region and ends when it leaves the frame. During its motion in the frame, the moving frame position is re-calculated every sample. The position is based on the frame boundary limits and the scaling ratio in moving frame units. To this basic formula, a correction term is added to correct a delay of 1-2 samples between the trigger from the sensor to the start of the operation. Due to the discretization of the SERCOS samples, this delay causes a gap in the position.

# Moving Frame

The input position is the current moving fame position. The input position (master source) is usually taken as an external source (external encoder), simulated axis or even another moving frame position.

If the master source is another moving frame (e.g., TCP-coordinates of any other kinematics with more than 2 axes), the offset/transformation between the tracking kinematics and the tracked kinematics is considered. In the implementation state of group as a master source (phase 3), the offset must be taken into consideration in **TRIGGER** or other alternative.

The tracking element and the moving frame must have the same kinematics system. It is impossible to track a different kinematics system.

The orientation is composed of two components: the master source (if there is one) and the orientation change due to movement. On the path, if the new position command is known, the new orientation can be calculated.

# Tracking Process

The tracking process consists of two phases: **approach** (where the robot position catches the moving frame position) and **track** (when both positions are synchronized (<***robot***>**.HERE** and <***moving frame***>**.HERE**)). The two phases are differentiated by the value of **ISMOVINGFRAMESYNCHRONIZED**. These are pure internal process phases over which you have no control. They are shown here to provide a better understanding of the implemented algorithms.

The robot motion is determined by ***modal*** values of velocity, acceleration and jerk both for rotation and translation (***ATRAN***, ***VTRAN***, ***DTRAN***, ***JTRAN***, ***VROT***, ***AROT***, ***DROT***, ***JROT***). These values are taken during moving frame assignment when <***element***>**.MASTERFRAME** is assigned. Later, changing these values does not affect the approaching process. For groups that do not have a kinematics model, usual kinematics values are used: ***VCRUISE***, ***ACC*** and ***JERK***. This phase is complete when both differences in velocity and position drop below certain thresholds. Higher values of these parameters assure faster approaches. Both robot's position and orientation coordinates are taken into account.

A system offers two different approach algorithms. One is the ***absolute tracking algorithm,*** and the other is the ***relative tracking algorithm***. Both are based on internal, closed prediction-PD position loop using position and velocity differences as success criteria. Use the absolute tracking algorithm in simple, slow-motion tracking with minimal initial distance of the robot from the tracking object. Use the relative tracking algorithm in all other cases. Besides differences in approaching algorithms there is also difference in the way both algorithms are used.

**Absolute Approach**

Absolute tracking algorithm (**slave = 3**)

The system automatically moves the robot position (<***robot***>**.SETPOINT**) toward the tracking object position (<***moving frame***>**.HERE**). Once synchronization is achieved, both positions are equal.

In this algorithm, two parameters influence the approaching behavior: <***moving frame***>**.FILTERFACTOR** and <***moving frame***>**.DAMPINGFACTOR**.

**NOTE**

***The requirement to enter the tracking phase is that the position error between the moving frame and the robot is less than half of the product between acceleration and the square of the sampling period (1/2 $AT^2$).***

**NOTE**

*FITERFACTOR* ***significantly influences the synchronization (approaching phase) behavior. Use small values in the beginning and increase them step-by-step until the desired results are obtained. High values of*** *FITERFACTOR* ***lead to robot instability!***

**Relative Approach**

Relative tracking algorithm (**slave = 5**)

The system moves the initial (before tracking) robot position (<***robot***>**.SETPOINT**) in the same direction as the tracking object position. Once synchronization is achieved, the robot moves synchronously on the same relative position to the moving object at the moment of approach start (**slave=0**).

It is your responsibility to add the appropriate movement and bring the robot position to the object. In this algorithm, the two parameters are used differently.

**FILTERFACTOR** defines the threshold needed to end the approach phase and switch to the track phase. As this parameter gets bigger, the approach phase lasts less time, but the transition moment becomes jerkier.

**DAMPINGFACTOR** tunes the prediction part of the algorithm. Larger values correspond to longer approach phases. Smaller values make the approach phase shorter.

**NOTE**

***Due to non-linear effects (kineamtics, velocity limitations, etc.), the algorithm becomes unstable with small*** *DAMPINGFACTOR* ***values. Low values of*** *DAMPINGFACTOR* ***lead to robot instability!***

## *CUSTOMER SUPPORT*

Danaher Motion products are available world-wide through an extensive authorized distributor network. These distributors offer literature, technical assistance, and a wide range of models off the shelf for the fastest possible delivery.

Danaher Motion sales engineers are conveniently located to provide prompt attention to customer needs. Call the nearest office for ordering and application information and assistance or for the address of the closest authorized distributor. If you do not know who your sales representative is, contact us at:

Danaher Motion
203A West Rock Road
Radford, VA 24141 USA
**Phone**: 1-540-633-3400
**Fax**: 1-540-639-4162
**Email**: customer.support@danahermotion.com
**Website:** www.DanaherMotion.com