SUTTER INSTRUMENT®

TRIO™ MPC-145 Series System (TRIO™ MPC-100)
External Control Quick Reference

Rev. 2.67k2 (20201201) (FW v2.62+)

Controlling the TRIO MPC-100 externally via computer is accomplished by sending commands over the USB interface between the computer and the USB connector on the rear panel of the TRIO MPC-100 controller/ROE. The USB device driver for Windows is downloadable from Sutter Instrument's web site (www.sutter.com). The TRIO MPC-100 requires Sutter Instrument's USB CDM (Combined Driver Model) Version 2.10.00 or higher. The CDM device driver consists of two device drivers: 1) USB device driver, and 2) VCP (Virtual COM Port) device driver. Install the USB device driver first, followed by the VCP device driver. The VCP device driver provides a serial RS-232 I/O interface between a Windows application and the TRIO MPC-100. Although the VCP device driver is optional, its installation is recommended even if it is not going to be used. Once installed, the VCP can be enabled or disabled.

The CDM device driver package provides two I/O methodologies over which communications with the controller over USB can be conducted: 1) USB Direct (D2XX mode), or 2) Serial RS-232 asynchronous via the VCP device driver (VCP mode). The first method requires that the VCP device driver not be installed, or if installed, that it be disabled. The second method requires that the VCP be installed and enabled.

Virtual COM Port (VCP) Serial Port Settings: The following table lists the required RS-232 serial settings for the COM port (COM3, COM5, etc.) generated by the installation or enabling of the VCP device driver.

Table 1. USB-VCP interface serial port settings.

Property	Setting
Data ("Baud") Rate (bits per second (bps))	57600
Data Bits	8
Stop Bits	1
Parity	None
Flow Control	None

The settings shown in the above table can be set in the device driver's properties (via the Device Manager if in Windows) and/or programmatically in your application.

Protocol and Handshaking: Command sequences do not have terminators. All commands return an ASCII CR (Carriage Return; 13 decimal, 0D hexadecimal) to indicate that the task associated with the command has completed. When the controller completes the task associated with a command, it sends ASCII CR back to the host computer indicating that it is ready to receive a new command. If a command returns data, the last byte returned is the task-completed indicator.

Command Sequence Formatting: Each command sequence consists of at least one byte, the first of which is the "command byte". Those commands that have parameters or arguments require a sequence of bytes that follow the command byte. No delimiters are used between command sequence arguments, and command sequence terminators are not used. Although most command bytes can be expressed as ASCII displayable/printable characters, the rest of a command sequence must generally be expressed as a sequence of unsigned byte values (0-255 decimal; 00 – FF hexadecimal, or 00000000 – 11111111 binary). Each byte in a command sequence transmitted to the controller must contain an unsigned binary value. Attempting to code command sequences as "strings" is not advisable. Any command data returned by the controller should be initially treated as a sequence of unsigned byte values upon reception. Groups of contiguous bytes can later be combined to form larger values, as appropriate (e.g., 2 bytes into 16-bit "word", or 4 bytes into a 32-bit "long" or "double word"). For the TRIO MPC-100, all axis position values (number of microsteps) are stored as "unsigned long" 32-bit positive-only values, and each is transmitted and received to and from the controller as four contiguous bytes.

Axis Position Command Parameters: All axis positional information is exchanged between the controller and the host computer in terms of microsteps. Conversion between microsteps and microns (micrometers) is the responsibility of the software running on the host computer (see *Microns/microsteps conversion* table for conversion factors).

Microsteps are stored as positive 32-bit values ("unsigned long" for C/C++, "uint32" for MATLAB, "U32" for LabVIEW, etc.). "Unsigned" means the value is always positive; negative values are not allowed. The positive-only values can also be stored in signed data type variables if necessary, in which case care must be taken to ensure that only positive values are exchanged with the controller (do not allow values that are less than 0).

The 32-bit value consists of four contiguous bytes, with a byte/bit-ordering format of Little Endian ("Intel") (most significant byte (MSB) in the first byte and least significant (LSB) in the last byte). If the platform on which your application is running is Little Endian, then no byte order reversal of axis position values is necessary. Examples of platforms using

Little Endian formatting include any system using an Intel/AMD processor (including Microsoft Windows and Apple Mac OS X).

If the platform on which your application is running is Big Endian (e.g., Motorola PowerPC CPU), then these 32-bit position values must have their bytes reverse-ordered after receiving from, or before sending to, the controller. Examples of Big-Endian platforms include many non-Intel-based systems, LabVIEW (regardless of operating system & CPU), and Java (programming language/environment). MATLAB and Python (script programming language) are examples of environments that adapt to the system on which each is running, so Little-Endian enforcement may be needed if running on a Big-Endian system. Some processors (e.g., ARM) can be configured for specific endianess.

Microsteps and Microns (Micrometers): All coordinates sent to and received from the controller are in microsteps. To convert between microsteps and microns (micrometers), use the following conversion factors (multipliers):

Table 2. Microns/microsteps conversion.

TRIO MPC-100 Controller with Device	From/To Units	Conversion Factor (multiplier)
MP-845/M, MP-845S/M, or **MP-245/M*** micromanipulator	μsteps → μm	0.09375
	μm → μsteps	10.66666666667
MP-285/M micromanipulator; **3DMS** or **MT-78** stage; **MOM** or **SOM** objective mover	μsteps → μm	0.125
	μm → μsteps	8

** DB25 to DB26HD adapter required for MP-245/M.*

For accuracy in your application, type these conversion factors as "double" (avoid using the "float" type as it lacks precision with large values). When converting to microsteps, type the result as a 32-bit "unsigned long" (C/C++), "uint32" (MATLAB), or "U32" (LabVIEW) integer (positive only) value. When converting to microns, type the result as a "double" (C/C++, MATLAB) or "DBL" (LabVIEW) 64-bit double-precision floating-point value.

Table 3. Ranges and bounds.

Device	Axis	Len. (mm)	Origin	Microns (Micrometers (μm))	Microsteps (μsteps)
MP-845/M, MP-845S/M, or MP-245/M* micromanipulator	X, Y, Z	25	BOT	0 – 25,000	0 – 266,667
MP-285/M micromanipulator; **3DMS** or **MT-78** stage; **MOM** or **SOM** objective mover	X, Y, Z	25	BOT	0 – 25,000	0 – 200,000

** DB25 to DB26HD adapter required for MP-245/M.*

NOTE: Origin is a physical position of travel that defines the center of the absolute position coordinate system (i.e., absolute position 0).
Physical Positions: BOT (Beginning Of Travel), COT (Center Of Travel), & EOT (End Of Travel).
In the TRIO MPC-100, the Origin is fixed at BOT.

NOTE: Travel length of each axis is automatically determined by end-of-travel sensor.

Travel Speed: The following table shows the travel speeds for supported devices using orthogonal move commands.

Table 4. Travel speeds.

Device	mm/sec or μm/ms	
	Single Axis	Dual Axis (x 1.4)
MP-845/M & MP-845S/M * micromanipulator	3	4.2
MP-285/M ** micromanipulator	5	7

** The same applies also to the MP-865/M & MP-245/M micromanipulators.*
*** The same applies also to the 3DMS & MPC-78 stages, MT-800-based translators, and SOM & MOM objective movers.*

Command Reference: The following table lists all the external-control commands for the TRIO MPC-100.

Table 5. TRIO MPC-100 external control commands

Command	Tx/-Delay/-Rx	Ver.	Total Bytes	Byte Offset (Len.)	Value			Alt-key-pad #	Ctrl-char	ASCII def./-char.	Description
					Dec.	Hex.	Binary				
Get Active Device & Firmware Version ('K')	**Tx**	All	1	0	75	4B	0100 1011	0075		**K**	Command. Returns the currently active device, along with firmware version number.
	Rx	All	4	0	1 – 2	01 – 02	0000 0001 – 0000 0010		^A – ^B	<SOH> – <STX>	Currently-active device (1 – 2 (A – B)).
				1							Major version number (e.g., if ver. = 2.62, then byte = 2)
				2							Minor version number (e.g., if ver. = 2.62, then byte = 62)
				3	13	0D	0000 1101		^M	<CR>	Completion indicator

Command	Tx/-Delay/-Rx	Ver.	Total Bytes	Byte Offset (Len.)	Value			Alt-key-pad #	Ctrl-char	ASCII def./-char.	Description
					Dec.	Hex.	Binary				
Change Active Device ('I')	**Tx**	All	2	0	73	49	0100 1001	0073		**I**	Command. Makes the specified device active, ensuring that all other commands are directed to that device.
				1	1-2	01 – 02	0000 0001 – 0000 0010	0001 – 0002	^A – ^B	<SOH> – <STX>	Manipulator to be made active (by value: 1(for A) or 2 (for B)
	Rx	All	2	0	1-2	01 – 02	0000 0001 – 0000 0010		^A – ^B	<SOH> – <STX>	Manipulator value specified
				1	13	0D	0000 1101		^M	<CR>	Completion indicator
Get Current Position and Angle ('c' or 'C')	**Tx**	All	1	0	99 or 67	63 or 43	0110 0011 or 0100 0011	0099 or 0043		'c' or 'C'	Command. Returns the current positions (μsteps) of X, Y, & Z axes and angle setting (degrees).
	Rx.	All	14		Current absolute positions of the X, Y, & Z axes, in microsteps, each consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value in Little-Endian bit order. See *Ranges* table for minimum and maximum values.						
				0 (4)							X pos. in μsteps
				4 (4)							Y pos. in μsteps
				8 (4)							Z pos. in μsteps
				12	0-90	00-5A	0000 0000 – 0101 1010			<NUL> – Z	Angle in degrees (0 – 90)
				13	13	0D	0000 1101		^M	<CR>	Completion indicator
Move to HOME Position ('h')	**Tx**	All	1	0	104	68	0110 1000	0104		'h'	Command. Moves to the position saved by the controller's HOME button. **X & Z** move first (angle determines order/simultaneity), and **Y** last.
	Rx	All	1	0	13	0D	0000 1101			<CR>	Completion indicator
Move to WORK Position ('w')	**Tx**	All	1	0	119	77	0111 0111	0119		'w'	Command. Moves to the position saved by the controller's WORK button. **Y** moves first, and **X & Z** last (angle determines order/-simultaneity).
	Rx	All	1	0	13	0D	0000 1101			<CR>	Completion indicator
Move to Specified "Home" Position ('H')	**Tx**	All	13	0	72	48	0100 1000	0072		'H'	Command. Move all 3 axes to specified position, moving **X & Z** (angle determines order/simultaneity), and **Y** last (see *Ranges* table).
					Target absolute positions of X, Y, & Z axes, in microsteps, each consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value (see *Ranges and bounds* table) in Little-Endian bit order.						
				1 (4)							X μsteps
				5 (4)							Y μsteps
				9 (4)							Z μsteps
	Rx	All	1	0	13	0D	0000 1101		^M	<CR>	Completion indicator

Command	Tx/-Delay/-Rx	Ver.	Total Bytes	Byte Offset (Len.)	Value Dec.	Value Hex.	Value Binary	Alt-key-pad #	Ctrl-char	ASCII def./-char.	Description
Move to Specified "Work" Position ('W')	**Tx**	All	13	0	87	57	0101 0111	0087		'W'	Command. Move all 3 axes to specified position, moving **Y** first, and **X & Z** last (angle determines order/simultaneity) (see *Ranges* table).
					Target absolute positions of X, Y, & Z axes, in microsteps, each consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value (see *Ranges and bounds* table) in Little-Endian bit order.						
				1 (4)							X μsteps
				5 (4)							Y μsteps
				9 (4)							Z μsteps
	Rx	All	1	0	13	0D	0000 1101		^M	<CR>	Completion indicator
Move in Straight Line to Specified Position at Specified Speed ('S')	**Tx**	All	14	0	83	53	0101 0011	0083		'S'	Command. Move all three axes simultaneously in a straight line to specified position (see *Ranges* table).
				1 (1)	15 – 0	0F – 00	0000 1111 – 0000 0000	0015 – 0000	^O – ^@		Speed (15 – 0 (fastest through slowest))
					Target absolute positions of X, Y, & Z axes, in microsteps, each consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value (see *Ranges and bounds* table) in Little-Endian bit order.						
				2 (4)							X μsteps
				6 (4)							Y μsteps
				10(4)							Z μsteps
	Rx	All	1	0	13	0D	0000 1101		^M	<CR>	Completion indicator
Interrupt Straight-Line Move (^C)	**Tx**	All	1	0	3	03	0000 0011	0003	^C	<ETX>	Command. Interrupts a move in progress (only for moves initiated by the "Straight-line" move ('S') command).
	Rx	All	1	0	13	0D	0000 1101			<CR>	Completion indicator
Move to specified X axis Position ('x' or 'X')	**Tx**	All	5	0	120 or 90	78 or 5A	0111 1000 or 0101 1010	0120 or 0090		'x' or 'X'	Command. Move X axis to specified position (see *Ranges* table).
				1 (4)	Target absolute position of X axis, in microsteps, consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value (see *Ranges and bounds* table) in Little-Endian bit order.						
	Rx	All	1	0	13	0D	0000 1101			<CR>	Completion indicator
Move to specified Y axis Position ('y' or 'Y')	**Tx**	All	5	0	121 or 91	79 or 5B	0111 1001 or 0101 1011	0121 or 0091		'y' or 'Y'	Command. Move Y axis to specified position (see *Ranges* table).
				1 (4)	Target absolute position of Y axis, in microsteps, consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value (see *Ranges and bounds* table) in Little-Endian bit order.						
	Rx	All	1	0	13	0D	0000 1101			<CR>	Completion indicator
Move to specified Z axis Position ('z' or 'Z')	**Tx**	All	5	0	122 or 92	7A or 5C	0111 1010 or 0101 1100	0122 or 0092		'z' or 'Z'	Command. Move Z-axis to specified position (see *Ranges* table).
				1 (4)	Target absolute position of Z axis, in microsteps, consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value (see *Ranges and bounds* table) in Little-Endian bit order.						
	Rx	All	1	0	13	0D	0000 1101			<CR>	Completion indicator

Command	Tx/-Rx	Ver.	Total Bytes	Byte Offset (Len.)	Value Dec.	Value Hex.	Value Binary	Alt-key-pad #	Ctrl-char	ASCII def./-char.	Description
Enter Angle ('A')	**Tx**	All	2	0	65	41	0100 0001	0065		'A'	Command. Sets the angle value, in degrees, to match the angle position of the rotary dovetail.
				1	0 - 90	00 - 5A	0000 0000 - 0101 1010	0000 - 0090		<NUL> - 'z'	Angle in degrees between 0 and 90. See *Angle Setting & Movement* note
	Rx	All	1	0	13	0D	0000 1101			<CR>	Completion indicator
Recalibrate ('R')	**Tx**	2.6	1	0	82	52	0101 0010	0082		'R'	Command. Causes the currently active manipulator to recalibrate.
	Rx	2.6	1	0	13	0D	0000 1101			<CR>	Completion indicator
Query Moving State ('q' or 'Q')	**Tx**	2.6	1	0	81 or 113	51 or 71	1001 0001 or 0111 0001	0081 or 0113		'q' or 'Q'	Command. Reports the state of movement for each device.
	Rx	2.6	3	0	0	00	0000 0000		^@	<NUL>	Device 1: Not moving
					1	01	0000 0001		^A	<SOH>	Device 1: Movement in progress
				1	0	00	0000 0000		^@	<NUL>	Device 2: Not moving
					1	01	0000 0001		^A	<SOH>	Device 2: Movement in progress
				2	13	0D	0000 1101			<CR>	Completion indicator

NOTES:

1. **Task-Complete Indicator:** All commands will send back to the computer the "Task-Complete Indicator" to signal the command and its associated function in controller is complete. The indicator consists of one (1) byte containing a value of 13 decimal (0D hexadecimal), and which represents an ASCII CR (Carriage Return).
2. **Intercommand Delay:** A short delay (usually around 2 ms) is recommended between commands (after sending a command sequence and before sending the next command).
3. **Clearing Send/Receive Buffers:** Clearing (purging) the transmit and receive buffers of the I/O port immediately before sending any command is recommended.
4. **Positions in Microsteps and Microns:** All positions sent to and received from the controller are in microsteps (μsteps). See *Microns/microsteps conversion* table) for conversion between μsteps and microns (micrometers (μm)).

   Declaring position variables in C/C++:

```
/* current position for X, Y, & Z */
unsigned long  cp_x_us, cp_y_us, cp_z_us;  /* microsteps */
double         cp_x_um, cp_y_um, cp_z_um;  /* microns */
/* specified (move-to) position for X, Y, & Z */
unsigned long  sp_x_us, sp_y_us, sp_z_us;  /* microsteps */
double         sp_x_um, sp_y_um, sp_z_um;  /* microns */
```

   Use the same convention for other position variables the application might need.

   Declaring the microsteps/microns conversion factors in C/C++:

```
/* conversion factors for the MP-845[S]/M based config. */
double us2umCF = 0.09375; /* microsteps to microns */
double um2usCF = 10.66666666667; /* microns to microsteps */
/* conversion factors for the MP-285/M based config. */
double us2umCF = 0.125; /* microsteps to microns */
double um2usCF = 8;             /* microns to microsteps */
```

   Converting between microsteps and microns in C/C++:

```
/* converting X axis current position */
cp_x_um = cp_x_us * us2umCF;  /* microsteps to microns */
cp_x_us = cp_x_um * um2usCF;  /* microns to microsteps */
```

   Do the same for Y and Z, and for any other position sets used in the application.

5. **Ranges and Bounds:** See Ranges and Bounds table for exact minimum and maximum values for each axis of each compatible device that can be connected. All move commands must include positive values only for positions – negative positions must never be specified. All positions are absolute as measured from the physical beginning of travel of a device's axis. In application programming, it is important that positional values be checked (>= 0 and <= max.) to ensure that a negative absolute position is never sent to the controller and that end of travel is not exceeded. All computational relative positioning must always resolve to accurate absolute positions.

   Declaring minimum and maximum absolute position variables in C/C++:

```
/* minimum and maximum positions for X, Y, & Z */
double min_x_um, min_y_um, min_z_um;  /* minimum microns */
double max_x_um, max_y_um, max_z_um;  /* maximum microns */
```

   Set minimum and maximum absolute positions for each axis – see Ranges & Bounds table.

```
/* initialize all minimum positions in microns*/
min_x_um = 0;
min_y_um = 0;
min_z_um = 0;
/* initialize all maximum positions in microns*/
/* MP-845[S]/M, MP-245[S]/M, MP-285/M, etc. */
max_x_um = 25000;
max_y_um = 25000;
max_z_um = 25000;
/* MP-865/M */
max_x_um = 50000;
max_y_um = 12500;
```

```
max_z_um = 25000;
```

6. **Absolute Positioning System Origin:** The Origin is set to a physical position of travel to define absolute position 0. The physical Origin position is fixed at beginning of travel (BOT). This means that all higher positions (towards end of travel (EOT)) are positive values; there are no lower positions and therefore no negative values are allowed.

7. **Absolute vs. Relative Positioning:** Current position ('c') and move commands always use absolute positions. All positions can be considered "relative" to the Origin (Position 0), but all are in fact absolute positions. Any position that is considered to be "relative" to the current position, whatever that might be, can be handled synthetically by external programming. However, care should be taken to ensure that all relative position calculations always result in correct positive absolute positions before initiating a move command.

   Declaring relative position variables in C/C++:

```
/* relative positions for X, Y, & Z */
double rp_x_um, rp_y_um, rp_z_um;  /* microns */
/* initialize all relative positions to 0 after
declaring them */
rp_x_um = rp_y_um = rp_z_um = 0;
```

   Enter any positive or negative value for each relative position (e.g., rp_x_um = 1000; rp_y_um = 500; rp_z_um = -200 … etc.

   For each axis, check to make sure that the new resultant absolute position (to which to move) is within bounds. Reset the relative position to 0 if not. If relative value is negative, its positivized value must not be greater than the current position. Otherwise, if positive, adding current position with relative position must not exceed the maximum position allowed. If out of bounds, resetting relative position to 0 allow the remaining conversions and movement to resolve without error.

```
/* check to make sure that relative X is within
bounds */
if ( ( rp_x_um < 0 && abs(rp_x_um) > cp_x_um ) ||
     (cp_x_um + rp_x_um > max_x_um) )  /* out of
bounds? */
    rp_x_um = 0; /* yes, so reset relative pos.
to 0 */
```

   Repeat the above bounds check for each of the remaining axes.

   For each axis, calculate new absolute position in microns and then convert to microsteps before issuing a move command.

```
/* convert X relative position to absolute posi-
tion */
sp_x_um = cp_x_um + rp_x_um; /* add relative pos.
to current pos. */
/* convert new absolute X position in microns to
microsteps */
sp_x_us = sp_x_um * um2usCF;
```

   Repeat for each of the remaining axes as required before issuing a move command.

8. **Position Value Typing:** All positions sent and received to and from the controller are in microsteps and consist of 32-bit integer values (four contiguous bytes). Position values in microsteps are always positive, so data type must be an "unsigned" integer that can hold 32 bits of data. Although each positional value is transmitted to, or received from, the controller as a sequence of four (4) contiguous bytes, for computer application computational and storage purposes each should be typed as an unsigned 32-bit integer ("unsigned long" in C/C++; "uint32" in MATLAB, "U32" in LabVIEW, etc.).

   Position values in microns (micrometers or μm) should be data typed as double-precision floating point variables ("double" in C/C++ and MATLAB, "DBL" in LabVIEW, etc.).

   Note that in Python, incorporating the optional NumPy package brings robust data typing like that used in C/C++ to your program, simplifying coding and adding positioning accuracy to the application.

9. **Position Value Bit Ordering:** All 32-bit position values transmitted to, and received from, the controller must be bit/byte-ordered in "Little Endian" format. This means that the least significant bit/byte is last (last to send and last to receive). Byte-order reversal may be required on some platforms. Microsoft Windows, Intel-based Apple Macintosh systems running Mac OS X, and most Intel/AMD processor-based Linux distributions handle byte storage in Little-Endian byte order so byte reordering is not necessary before converting to/from 32-bit "long" values. LabVIEW always handles "byte strings" in "Big Endian" byte order irrespective of operating system and CPU, requiring that the four bytes containing a microsteps value be reverse ordered before/after conversion to/from a multibyte type value (I32, U32, etc.). MATLAB automatically adjusts the endianess of multibyte storage entities to that of the system on which it is running, so explicit byte reordering is generally unnecessary unless the underlying platform is Big Endian. If your development platform does not have built-in Little/Big Endian conversion functions, bit reordering can be accomplished by first swapping positions of the two bytes in each 16-bit half of the 32-bit value, and then swap positions of the two halves. This method efficiently and quickly changes the bit ordering of any multibyte value between the two Endian formats (if Big Endian, it becomes Little Endian, and if Little Endian, it becomes then Big Endian).

10. **Travel Lengths and Durations:** "Move" commands might have short to long distances of travel. If not polling for return data, an appropriate delay should be inserted between the sending of the command sequence and reception of return data so that the next command is sent only after the move is complete. This delay can be auto calculated by determining the distance of travel (difference between current and target positions) and rate of travel. This delay is not needed if polling for return data. In either case, however, an appropriate timeout must be set for the reception of data so that the I/O does not time out before the move is made and/or the delay expires.

11. **Movement Speeds:** All move commands cause movement to occur at a maximum rate of 3,000 microns/second (MP-845/M based) or 5,000 microns/second (MP-285/M based), except for the "Straight-Line Move 'S' command which can be specified with one of sixteen speeds. Actual speed for the "Straight-Line Move 'S' command can be determined with the following formula: (max_speed / 16) * (sp +1), where "max_speed" is the maximum speed in microns/second (3,000 or 5,000) and "sp" is the speed level 0 (slowest) through 15 (fastest). For mm/second or microns/millisecond, multiply result by 0.001.

Table 6. Straight-Line Move 'S' Command Speeds for MP-845/M-based configuration.

Speed Setting	mm/sec or μm/ms	μm/sec or nm/ms	nm/sec	in/sec or mil/ms	% of Max.
15	3.0000	3000.0	3000000	0.1181102360	100.00%
14	2.8125	2812.5	2812500	0.110728346	93.75%
13	2.6250	2625.0	2625000	0.103346457	87.50%
12	2.4375	2437.5	2437500	0.095964567	81.25%
11	2.2500	2250.0	2250000	0.088582677	75.00%
10	2.0625	2062.5	2062500	0.081200787	68.75%
9	1.8750	1875.0	1875000	0.073818898	62.50%
8	1.6875	1687.5	1687500	0.066437008	56.25%
7	1.5000	1500.0	1500000	0.059055118	50.00%
6	1.3125	1312.5	1312500	0.051673228	43.75%
5	1.1250	1125.0	1125000	0.044291339	37.50%
4	0.9375	937.5	937500	0.036909449	31.25%
3	0.7500	750.0	750000	0.029527559	25.00%

Speed Setting	mm/sec or μm/ms	μm/sec or nm/ms	nm/sec	in/sec or mil/ms	% of Max.
2	0.5625	562.50	562500	0.022145669	18.75%
1	0.3750	375.00	375000	0.014763780	12.50%
0	0.1875	187.50	187500	0.007381890	6.25%

Table 7. Straight-Line Move 'S' Command Speeds for MP-285/M-based configuration.

Speed Setting	mm/sec or μm/ms	μm/sec or nm/ms	nm/sec	in/sec or mil/ms	% of Max.
15	5.0000	5000.0	5000000	0.196850394	100.00%
14	4.6875	4687.5	4687500	0.184547244	93.75%
13	4.3750	4375.0	4375000	0.172244094	87.50%
12	4.0625	4062.5	4062500	0.159940945	81.25%
11	3.7500	3750.0	3750000	0.147637795	75.00%
10	3.4375	3437.5	3437500	0.135334646	68.75%
9	3.1250	3125.0	3125000	0.123031496	62.50%
8	2.8125	2812.5	2812500	0.110728346	56.25%
7	2.5000	2500.0	2500000	0.098425197	50.00%
6	2.1875	2187.5	2187500	0.086122047	43.75%
5	1.8750	1875.0	1875000	0.073818898	37.50%
4	1.5625	1562.5	1562500	0.061515748	31.25%
3	1.2500	1250.0	1250000	0.049212598	25.00%
2	0.9375	0937.5	937500	0.036909449	18.75%
1	0.6250	0625.0	625000	0.024606299	12.50%
0	0.3125	312.50	312500	0.012303150	6.25%

12. **Move Interruption and Concurrent Dual Manipulator Movements:** A command should be sent to the controller for a manipulator only after the task of any previous command for that same manipulator is complete (i.e., the task-completion terminator (CR) is returned associated with the manipulator). One exception is the "Interrupt Move" (^C) command, which can be issued while an 'S' command-initiated move for a manipulator is still in progress. While a move is in progress for a manipulator (e.g., A), another move command sequence can be issued for the other manipulator (i.e., B).

13. **Angle Setting & Movement:** Although the set angle command allows for a range of 0° to 90°, the effective range that allows full movement is 1° to 89° (>0° and <90°). If 0° or 90°, Z or X axis fails to move, causing single- and multi-axis movement commands to fail. The ideal range for smooth movement is 10° to 80°. Factory default is 30°.

14. **Extracting the MPC-100 Firmware Version Number:** The firmware version number returned by the 'K' command is encoded in two bytes, with major version byte first, followed by minor version byte. For example, if the complete version is 2.62, then the bytes at offsets 1 and 2 will show (in hexadecimal) as 0x02 0x3E (ret[1] and ret[2] as shown in the following code snippets). The following code shows how to extract and convert the version into usable forms for later comparison without altering the original command return data (written in C/C++ and is easily portable to Python, Java, C#, MATLAB script, etc.).

```
/* "ret" is the array of bytes containing
the 'K' command's return data at offsets 1 & 2 */
/* define variables */
unsigned char minver, majver;
int majminver;
float version;
```

Major version number as an integer (e.g., 2:

```
majver = ret[1];  /* get major ver. */
```

Minor version number as an integer (e.g., 62:

```
minver = ret[2]; /* get minor ver. */
```

Complete (thousands) version as an integer (e.g., 262):

```
majminver = majver * 100 + minver;
```

Complete version as a floating-point number (e.g., 2.62):

```
version = majminver * .01;
```

NOTES:

NOTES: