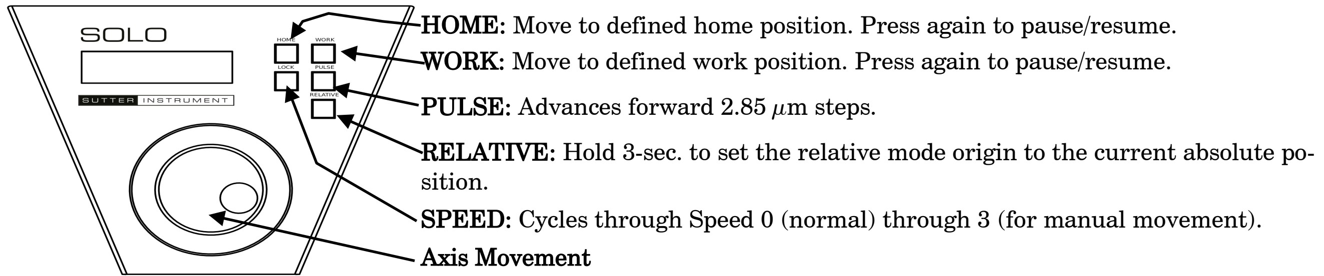
SUTTER INSTRUMENT®

SOLO Quick Reference

Rev. 1.08b (20220613) (FW v2.55+))

Manual Operation



Setting Home/Work Pos. & Relative Mode Origin Pos.: To set position, hold down HOME, WORK, & RELATIVE buttons for 3 seconds until beep sounds.

Screen-color mode indications: Green = Position status; Red = Movement in progress (knob disabled).

Movement Knob Disabling: Movement knob is disabled during movement to Home, Work, or when initiated by external movement command.

Configuration

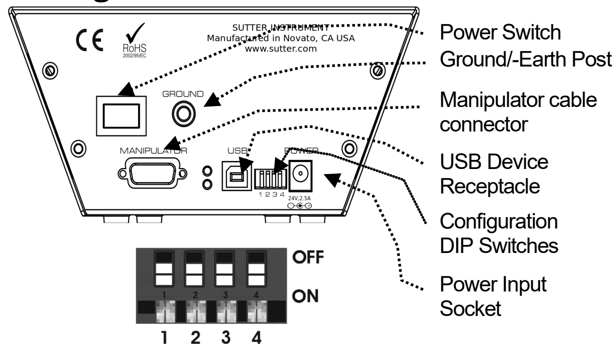


Table 1. Configuration Switches 1 – 4.

Sw #	Definition	State	Setting	Position
1	Knob Rotation for Forward (+) Movement	Counter	OFF	UP
		Clockwise	**ON***	**DOWN***
2	Single axis travel length (25 or 50 mm)	50mm	OFF	UP
		25mm	**ON***	**DOWN***
3	Electromechanical device compatibility	MP-285/M	OFF	UP
		SOLO/M	**ON***	**DOWN***
4	Calibration Homing on Power On	Disabled: **	OFF	UP
		Enabled.	**ON***	**DOWN***

* *Normal operation (factory default).*

** *Switch 4 OFF (up) to retain power-off position.*

External Control

Controlling the SOLO 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 SOLO controller/ROE. The USB device driver for Windows is downloadable from Sutter Instrument's web site (www.sutter.com). The SOLO 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 SOLO. 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 2. 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.

COPYRIGHT © 2022 SUTTER INSTRUMENT COMPANY. ALL RIGHTS RESERVED

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 SOLO, 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 ("long" (or optionally, "signed long"), or "unsigned long" for C/C++; "I32" or "U32" for LabVIEW). "Unsigned" means the value is always positive; negative values are not allowed. The positive-only values can also be stored in signed type variables, in which case care must be taken to ensure that only positive values are exchanged with the controller.

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 <u>after</u> receiving from, or <u>before</u> 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 3. Microns/microsteps conversion

SOLO Controller with Device	From/To Units	Conversion Factor (multiplier)
SOLO-25/M or **SOLO-50/M,** or single axis of a **TRIO** series or **QUAD/M** electromechanical	µsteps → µm	0.09375
	µm → µsteps	10.66666666667
Single axis of an **MP-285/M** micromanipulator or derived electromechanical	µsteps → µm	0.125
	µm → µsteps	8

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.

Ranges and Bounds:

Table 4. Ranges and bounds.

Device	Axis	Len. (mm)	Origin	Microns	Microsteps
SOLO-25/M	(any)	25	BOT	0 – 25,000	0 – 266,667
SOLO-50/M	(any)	50	BOT	0 – 50,000	0 – 533,334

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

Table 5. Travel speeds.

Device	mm/sec or µm/ms
SOLO-25/M or **SOLO-50/M,** or single axis of a **TRIO** series or **QUAD/M** electromechanical *	3
Single axis of an **MP-285/M** micromanipulator or derived electromechanical **	5

Command Reference: The following table lists all the external-control commands for the SOLO.

Table 6. SOLO external control commands

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
Get Current Position ('c' or 'C')	**Tx**	All	1	0	99 or 67	63 or 43	0110 0011 or 0100 0011	0099 or 0043		'c' or 'C'	Returns the current position (μsteps) of the single axis
	Rx	All	5	0-3	Current absolute position of the single axis, in microsteps, consisting of 4 contiguous bytes representing a single 32-bit unsigned (positive) integer value in Little-Endian bit order.						
				4	13	0D	0000 1101		^M	<CR>	Completion indicator
Move to HOME Position ('h')	**Tx**		1	0	104	68	0110 1000	0104		'h'	Moves to the position saved for the controller's HOME button
	Rx		1	0	13	0D	0000 1101		^M	<CR>	Completion indicator
Move to WORK Position ('w')	**Tx**		1	0	119	77	0111 0111	0119		'w'	Moves to the position saved for the controller's WORK button
	Rx		1	0	13	0D	0000 1101		^M	<CR>	Completion indicator
Move to Specified "Home" Position ('H')	**Tx**	All	5	0	72	48	0100 1000	0072		'H'	Move single axis to specified position (see *Ranges* table)
				1-4	Target absolute position, 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		^M	<CR>	Completion indicator
Move to Specified "Work" Position ('W')	**Tx**	All	5	0	87	57	0101 0111	0087		'W'	Move single axis to specified position (see *Ranges* table)
				1-4	Target absolute position, 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		^M	<CR>	Completion indicator
Move to specified X axis Position ('x' or 'X')	**Tx**		5	0	120 or 88	78 or 58	0111 1000 or 0101 1000	0120 or 0088		'x' or 'X'	Move X axis to specified position (see *Ranges* table)
				1 - 4	Target absolute position, 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		1	0	13	0D	0000 1101		^M	<CR>	Completion indicator
Set Velocity ('v')	**Tx**	2.55 +	3	0	118	76	0111 0110	0118		'v'	Sets the velocity for all external-control movement commands
				1 (2)	Speed factor consisting of 2 contiguous bytes representing a single 16-bit unsigned (positive) integer value in Little-Endian bit order ranging from 0 (fastest) to 65,535 (slowest)						
	Rx	All	1	0	13	0D	0000 1101		^M	<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 */
   unsigned long  cp_x_us; /* microsteps */
   double         cp_x_um; /* microns */
   /* specified (move-to) position for X */
   unsigned long  sp_x_us; /* microsteps */
   double         sp_x_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 SOLO-xx/M based config. */
   double us2umCF = 0.09375;        /* microsteps to microns */
   double um2usCF = 10.66666666667; /* 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 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;  /* minimum microns */
   double max_x_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;
   /* initialize all maximum positions in microns*/
   /* SOLO-25/M */
   max_x_um = 25000;
   /* SOLO-50/M */
   max_x_um = 50000;
   ```

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, & D */
   double rp_x_um;  /* microns */
   /* initialize all relative positions to 0 after declaring them */
   rp_x_um = 0;
   ```

   Enter any positive or negative value for each relative position (e.g., rp_x_um = 1000).

   For the single 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 */
   ```

   Calculate new absolute position in microns and then convert to microsteps before issuing a move command.
   ```
   /* convert X relative position to absolute position */
   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;
   ```

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 Speed:** All move commands cause movement to occur at a maximum rate of 3,000 microns/second. With firmware v2.55 and above, speed can be controlled for all externally controlled movements using the 'v' command followed by a 16-bit value of 0 (fastest) to 65,535 (slowest) (Little-Endian bit order).

NOTES:

NOTES: