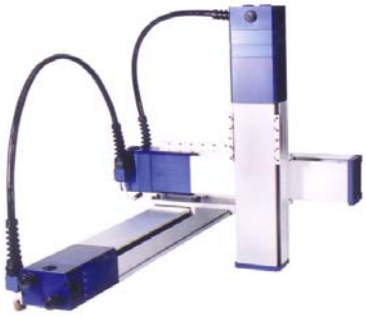
Smart Linear Actuator User's Guide

Table Of Contents

Smart Linear Actuator

Operating Software 1.9 User Manual



SLA Operating Software (SLA OS) is used to program DE-STA-CO Automation's Smart Linear Actuator electric slides. The SLA OS User manual explains the many features available in the software..

New features summary

Following is a summary of the main features in each of the release.

v1.9

- New Features
    - SLA OS is now compatible with the upcoming line of eCylinder (EC65, EC90) and eRotary (ER21, ER75) product line. As an end user, all the investment in learning the SLA OS software is preserved by the backward compatible changes. The virtual mode is also upgraded to gracefully handle the new product line.
    - When starting SLA OS application, there is an additional drop down menu for choosing an application profile. Current choices are
        1. Complete SLA OS (default) - this will allow to leverage full functionality of the software.
        2. eCylinder/eRotary Programming - this will result in a simple user interface geared towards BCD programming for eCylinder/eRotary.
    - Improved BCD programming has following new features
        1. A new BCD wizard provides choice of either creating a BCD program from scratch or from the existing collection of points.
        2. Adding, updating and deleting positions within the BCD programming screen itself
        3. Now it is possible to assign different speed, acceleration for each BCD point. Note that speed and acceleration are specified as percentage of maximum limits.
        4. Newly introduced Relative position mode (in addition to the present Absolute position mode) can be assigned to a point which results in incremental motion. This could be used in an application where PLC signal can move a slide/rotary by fixed distance/angle respectively.
    - Dynamic path creation capability for MMP programs can now also be used for creating curvilinear (arc) paths by assigning the dynamic path the pathType value of 2 (1 = linear (default), 2 = curvilinear) and adding 'start', 'center', 'end' and 'direction' points.
- SLA Commands
    - Included Calibration as a value that can be obtained using sla.MotorStatus command. This can be used to determine if a particular motor has been calibrated (homed) and to alert operator before proceeding to calibrate it.
- Miscellaneous
    - A new submenu from Diagnostics icon in toolbar allows releasing serial connection without closing the whole application. When needed, the connection can be reestablished by running the Diagnostics again.
    - Teach pendant has been enhanced to change the speed and acceleration as percentage of maximum values for each motor.
    - Dataset window now has status bar that shows useful information about datasets, paths and points. It also shows elapsed time in seconds for running a path to help in optimizing cycle times.
    - Unit Conversion calculator is enhanced to work with new eRotary motors for converting angles in degrees to Smart Motor Units.
    - Single axis representation is improved in Dataset window which also allows creating BCD points for one axis system by clicking on the graphical area.

v1.8

- New Features
    - Added Wizards for automatically creating MMP programs for common tasks such as palletizing and glue dispensing. In the subsequent releases more wizards will be added to

simplify all aspects of programming including template support where special templates will be created for speeding up everyday common programming tasks.
  - Enhanced Dataset Path creation by adding functionality to copy, paste and insert a point anywhere within a path. Apart from the existing 'create point' and 'update point' modes, now there is an additional 'insert point' which can be enabled by pressing CTRL and clicking on any point. In the insert mode, a new point can be inserted anywhere within a path by selecting a point and creating a point in usual manner. To copy and paste a point, use CTRL+C (for copy) and CTRL+V (for paste) commands.
  - A new path that starts at the end of another path can be created by selecting the lead path and creating the new path. This works even when the lead path is an arc. This functionality is very useful while creating elaborate appended paths (e.g. for dispensing application).
  - New dynamic path creation capability of MMP programs allows constructing user defined linear paths at runtime. This is useful for creating paths based on user input or from values stored in external file.
  - Extended the arc manipulation CTRL command to adjust the end point of an arc to lie on the arc. To access this feature, select the end point, press CTRL and click anywhere on the graphical area. This will modify the end point's values so that it is on the arc now. This feature is useful for obtaining the start point for the next path in the appended chain of paths.
  - Created SMP debugging capability by having SMP programs use PRINT statements to print on the motor terminal. These statements can report the existing variable values as well as text messages for debug help.
  - Added capability to interface with DVT Vision System. The data obtained from the vision system can be used to drive the SLA system.
  - SLA OS software now comes bundled with plenty of useful sample programs and their associated databases to run out of box. These programs (both SMP and MMP) use recommended programming practices and can be extended to create custom programs. The programs are located in the samples directory under the installation directory.
  - Created additional log levels to use from within MMP programs. Currently supported log levels are
    - **0 (new)** can be used for logging messages which are not saved in the file.
    - **1** can be used for logging information messages. These messages are also saved in the log file.
    - **2** can be used for logging warning messages. These messages are also saved in the log file.
    - **3** can be used for logging error messages. These messages are also saved in the log file.
    - **4 (new)** can be used for logging messages which are displayed in a separate pop up window with bigger fonts. This is useful for monitoring the program's progress. These messages are not saved in the log file.
- EXCEL Commands
  - Added new excel commands (excel.ReadFile, excel.GetValue) for reading Microsoft Excel files without a need for installing Excel on the computer. This allows additional means to control the behavior of a program depending on the data present in the file. Note that the first line of the file is ignored and the second row is treated as row one. Also, if a column has mixed types of data (e.g. numbers and text), this method of retrieval can return Null for the minority types of data and some registry settings need to be modified to get the correct results. Please see http://support.microsoft.com/kb/194124/EN-US/ or contact technical support for additional help.
- SLA Commands
  - Enhanced sla.WaitForStop command to raise an error if motor is stopped due to fault (e.g. limit switch or position error).

    - Included MaxCurrent and MaxPositionError as two additional values that can be obtained and set using sla.MotorStatus command. This allows to control maximum torque that can be exerted and adjust the threshold when position error is flagged.
- VISION Commands
    - Added new vision commands (vision.connectDVT, vision.ProcessDVT, vision.GetDVTValue) for interfacing with DVT vision system. This extends the existing support for Cognex to include two of the most widely used vision systems in the industry.
- Miscellaneous
    - Enhanced SMP Development Environment with ability to open multiple SMP programs simultaneously, multi level undo/redo facility, enhanced syntax coloring, selecting fonts and additional toolbar buttons.
    - Ability to select COM1 to COM9 ports for SLA OS operation. Earlier it was restricted up to COM4. This feature is useful when using USB to Serial converter which can create higher numbered serial ports.
    - Enhanced calculator with ability to copy result and handle invalid inputs.

v1.7.1

- New Features
    - Added an ability to create arc paths from the start, end and any other point on the arc. This augments the existing capability where start, end and the center points of an arc are required. This is very useful when the exact dimensions are not available. In this case, the three points can be updated by moving the slides to the three points of an actual part. To access this feature, first update the start and the end points. Then to calculate the center point automatically, press CTRL and any point on the arc. This will create the desired center point.

v1.7

- New Features
    - Added Virtual Motors Simulation (VMS) functionality to run the software in simulation mode.
        - Very useful when the actual hardware is not yet available or for training without the need of actual hardware.
        - Up to six virtual motors can be selected and configured individually.
        - Visual and audio simulation along with the tracing functionality of dataset window can be used to visualize the physical system behavior.
        - Configure simple rules for input switches depending on the state of an output switch to simulate physical IO behavior.
        - Input switches can be operated from the Motor Monitor screen.
        - Currently the MMP programming mode is fully supported. The SMP and BCD programs can be written and debugged but can't be run at this time.
    - Added time conversion in the Unit Conversion Calculator. This can be used to get time in SMU units when writing SMP program.
- SERIAL Commands
    - Added new serial commands (serial.OpenPort, serial.WritePort, serial.ReadPort and serial.ClosePort) for reading and writing from a MMP program using serial.* convention.
- Miscellaneous
    - Added validation for point, path and dataset naming convention. Now it checks that these names don't contain space or illegal characters.
    - Improved documentation with Macromedia Flash tutorials. Now it is even easier to learn the software by following the video tutorials accessible from the help menu.

v1.6

- New Features
  - Added Position Data Import for AutoCad Exchange Format (DXF), Microsoft Excel (XLS) and Comma Separated Values (CSV) files.
  - Added capability to interface with Cognex Vision System. The data obtained from the vision system can be used to drive the SLA system.
  - Added Unit Conversion Calculator for converting position, velocity and acceleration to various units. The calculator takes into account the pitch and encoder counts of a motor for conversion.
  - BCD programming now supports in place editing, on-the-fly program generation and synchronization with the dataset view.
- SLA Commands
  - Modified sla.WaitForStop with timeOut parameter which allows to wait only for specified time period.
- Vision Commands
  - Introduced vision.ConnectCognex, vision.ProcessCognex and vision.GetCognexValue commands to support Cognex vision interface
- Miscellaneous
  - BCD programming can now contain 100 positions from earlier 50 positions limitation.
  - Added scroll bar for Teach Pendant to accommodate SLA configurations involving high number of motors.
  - Added client specific software customization feature to provide enhanced level of technical service.
  - Enhanced syntax coloring of SMP programs, added Save As functionality to save SMP program with different name, added functionality to detect changed SMP program and confirm saving when exiting.
  - 'Auto Brake' button is automatically disabled when servo is on.

v1.5.1

- SLA Commands
  - Added the command sla.RunDiagnostics to reset the serial connection in the event of power cycle e.g. when an E-Stop button is pressed.
  - Added decelerate option to sla.WaitForStop and sla.StopMotion commands to control the velocity profile while stopping. Now by default, it decelerates to stop in contrast to sudden stop earlier.
- HMI Commands
  - Added decelerate option to hmi.WaitForStop command to control the velocity profile while stopping. Now by default, it decelerates to stop in contrast to sudden stop earlier.
- Miscellaneous
  - Now teach pendant takes into account the selected speed and acceleration when desired position is entered into the position textbox and return key is pressed. Earlier it used the default velocity and acceleration.

v1.5

- SLA Commands
  - Added sla.StopMotion, sla.DoPositionMove, sla.DoVelocityMove commands
  - Added additional status values (Bd, Bs, Bu) that can be accessed using sla.MotorStatus
  - Modified sla.WaitForStop command. The modified command has the capability to block the execution of program till an input signal attains the specified value or the motion stops, whichever occurs first. The modified command is backward compatible.

- Modified sla.CheckSwitch command. The modified command can return the status of either input or output switch. Earlier it used to return the status of only input switch. The modified command is backward compatible.
- Command sla.WaitForSwitch now returns True/False depending on success or failure.
- Removed redundant sla.MoveToPath command. The same operation can be carried out using sla.BeginningPathPoint which returns the first point on the path and then using DoLine to move to the point.

Removal of the command can result in backward incompatibility. See the following example about suggestions for fixing the code to work with this and all future releases.

```
        Replace

                sla.MoveToPath(MyDataset.path1, 100, 1000)

        with

        sla.DoLine(sla.BeginningPathPoint(MyDataset.path1),
100, 1000)
```

- Removed redundant sla.ShiftAxisToPoint command.The same operation can be carried out using sla.ShiftAxis command (see below).

Removal of the command can result in backward incompatibility. See the following example about suggestions for fixing the code to work with this and all future releases.

```
        Replace

                sla.ShiftAxisToPoint(MyDataset.point1,
isChainedVariable)

        with

                Dim pointVariable as Point

                Set pointVariable = MyDataset.point1

                sla.ShiftAxis(pointVariable.x,
pointVariable.y, pointVariable.z, isChainedVariable)
```

- Removed sla.DoMotion command. Replaced with the new sla.DoVelocityMove command.

Removal of the command can result in backward incompatibility. See the following example about suggestions for fixing the code to work with this and all future releases.

Replace

```
sla.DoMotion(motorIndex, direction, speed,
acceleration, monitorMotorIndex, monitorInputSwitch ,
monitorInputSwitchStatus )
```

with

```
sla.DoVelocityMove(motorIndex, direction,
speed, acceleration)

sla.WaitForStop(motorIndex,
monitorMotorIndex, monitorInputSwitch ,
monitorInputSwitchStatus)
```

- HMI Commands
  - Added hmi.WaitForStop command which is similar to sla.WaitForStop command.
- Miscellaneous
  - Help can be accessed from Help menu or clicking 'F1' function key.
  - MMP Development environment saves and restores the selected fonts.
  - Fixed bug regarding zero grid dimensions which freezes the SLA OS software.
  - Provided new button on the monitor screen to turn on/off all Outputs for testing purpose.

v1.4.1

- SLA Commands
  - Extended the sla.SetServoOff command to optionally turn off automatic brakes. This can be used to easily move the slides around with hands. Since turning the servo off and the brakes at the same time can lead to undesired motions (e.g. z slide falling down due to gravity), removed the option of applying the command to all motors. Now the command has to be applied to individual motor explicitly (see below).

Removal of the option to apply the command to all motors (indicated by argument '0') can result in backward incompatibility. See the following example about suggestions for fixing the code to work with this and all future releases.

```
Replace
        sla.setServoOff(0)
with
        sla.setServoOff(1)
        sla.setServoOff(2)
        sla.setServoOff(3)
```

  - Extended the sla.Calibrate command to optionally provide user specified speed and acceleration to control the speed of homing.
- HMI Commands
  - New command hmi.SaveIOs available through sla commands to make it possible to programmatically persist the Modbus data
  - New command hmi.WriteRegister and hmi.ReadRegister deals with individual 16 bits Modbus registers.

v1.4

- SLA Commands
  - Added a new command sla.WaitForStop to wait for all motors to stop moving.
  - Enhanced sla.ElapsedTime function to also allow resetting the timer
  - Enhanced sla.DoMotion command to specify an input to monitor and stop the motion when the input has the specified state (1/0).
- DB Status - Since the new version allows any database to be used (chosen during the login), it is necessary to provide the information about which database is currently in used. Created and added the DB icon in the status bar for this purpose.
- Bug fixes
  - When an invalid IO is specified, now it gives more meaningful message instead of 'No response received' earlier
  - Introducing Point class in the Basic environment even if no points are defined.
  - Fixed validation error for RotateAxis which can specify rotation around z-axis even when z-axis is not present.

v1.3

- Multiple Configuration
  - Ability to switch between different databases from Login window
  - Ability to configure batchDatabase from Configuration window
  - Ability to save current database under different name from Logout window
- Status Variables - The new command in SLA Basic environment makes available all the status variables for each of the motor. This gives flexibility in programming as well as monitoring.
- Persistent Modbus Data - Additional database table holds the value of Modbus data. This value is restored when the application restarts. The database migration feature is updated to update any old database to automatically detect and update with the new table structure.

v1.2

- Enhanced Programming Environment - The enhanced environment for writing Basic programs offers most of the standard text editor (Save As, Find/Replace, ...) functionalities.
- Uninstall - Now it is possible to uninstall the application through Start menu. Earlier user had to go to Control Panel and Add/Remove programs to remove the application.
- Better File Management - Enhanced install detects the existing database files and doesn't overwrite them. Similarly, enhanced uninstall doesn't uninstall the important files.

v1.1

- HMI - Incorporating industry standard Modbus protocol in the application to offer HMI device compatibility for industrial applications.

v1.0

- Multitasking - This allows multiple *.mmp (multi motor programs) to run at the same time. The immediate use of this feature is to use the supplied 'startup.mmp' program to stop/start any mmp program by switching on corresponding input.
- Three dimensional motion - The linear as well as curvilinear motion is extended to three dimensions. What this allows is to draw arcs/circles on any planes in space. Also, the addition of a 'pitch/rotations' allow a helical movement in any direction in space.
- Appending arbitrary paths - This allows making a very complex path made up of various linear and curvilinear motions and running the slides with a smooth motion along the complete path.
- Path calculation caching - Since calculating the points to send to motor for a given path is one of the most CPU intensive task, a new caching mechanism is integrated. This allows to reuse the calculations repeatedly, thus speeding up the program tremendously.
- Multi-pitch capability - The software can handle any combination of SLA slides (e.g. X-axis SLA-150, Y-axis SLA-120 and Z-axis SLA-90). This allows for much more flexible configuration. The software compensates for the difference in pitch ratios among the slides.
- State management - When a program crashes or computer loses power, we still need to maintain state across these events. The new saveState/getState/removeState sla functions provide this capability. The common use is to 'start from where left off' functionality. e.g. in a N1 X N2 grid if we are drilling/inking and program crashes when it has completed nth grid point, next time when the program starts, we would want to repeat the process for the first n points and start at n+1 th point. The state management makes this possible. Also, note that it has been enhanced so that it works across threads which allows multiple threads to communicate among themselves.
- Pre-defined MMP function library - There is a provision of providing pre-written functions (slaFunctions.mmp) which can be used in any mmp program. We also provide another file userFunctions.mmp which can be customized for the end user for use in their applications.
- Database migration - The database that holds the SLA data has a certain structure when the application is shipped. However, with the new database migration capability, when new features are added or there is a need to change the existing database structure (schema), it can be handled without having customer intervention.
- Profiling for time - With the introduction of StartTimer and ElapsedTimer functions in sla, it is very simple to keep track of more than one timers to do profiling (e.g. cycle time) study.
- Robust BCD auto generated programs - The new template for generating BCD programs has been greatly improved to make it a 'one-click-process' to create a BCD program.

Contacting

SITE LICENSE AGREEMENT

IMPORTANT-READ CAREFULLY: This DE-STA-CO Automation End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and DE-STA-CO Automation ("DE-STA-CO Automation") for the DE-STA-CO Automation Licensed Product identified as DE-STA-CO Automation SLA operating software ("LICENSED PRODUCT"). By installing, copying, or otherwise using the LICENSED PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, you may not use the LICENSED PRODUCT.

In consideration of the mutual covenants that follow, Licensor and Licensee agree:

1. Definitions

a. *Licensed Product.* The term “Licensed Product” means the DE-STA-CO Automation SLA operating software.

b. *Authorized User(s).* The term “Authorized User(s)” means any current employee of Licensee.

c. *Representative Authorized User(’s),* The term “Representative Authorized User(s)” means any Authorized User under the terms of this Agreement who secures a copy of the Licensed Product for use on a DE-STA-CO Automation SLA modular automation system. A Representative Authorized User is charged with the custody, supervision, control, and security of the copy of the Licensed Product which he or she receives from the Licensee. For purposes of this Agreement, a person acting as a Representative Authorized User is acting on behalf of Licensee.

d. *Local Network System(s).* The term “Local Network System(s)” means multiple, interactive user terminals connected to a single-processing or a multi-processing microcomputing unit whereby the user of an interactive terminal does not have physical access to the physical storage medium containing a copy of the Licensed Product.

e. *Free-Standing Workstation(s).* The term "Free-Standing Workstation(s)" means a self-contained microcomputing unit which is owned or leased by Licensee for the exclusive use of Licensee's employees and for which Licensee has provided a copy of the Licensed Product.

2. License

In accordance with the terms of this Agreement, Licensor grants to Licensee, and Licensee accepts from Licensor, a nonexclusive and nontransferable license to use the Licensed Product on the Local Network Systems and Free-Standing Workstations owned, leased, or operated by Licensee for use only by Authorized Users. The use of the Licensed Product and related documentation is expressly limited to use on the DE-STA-CO Automation SLA modular automation system.

3. Ownership of Licensed Product

Licensor represents that it is, and on the date of delivery of Licensed Product will be, the sole owner and copyright holder of Licensed Product; that it has, and on the date of the delivery of the Licensed Product will have, the full right and authority to grant this license and that neither this license nor performance under this Agreement does or shall conflict with any other agreement or obligation to which Licensor is a party or by which it is bound.

4. Title to and Rights in Licensed Product

a. *Proprietary Rights*. The Licensed Product and updates of the Licensed Product are proprietary to Licensor, and title to them remains in Licensor. All applicable common law and statutory rights in the Licensed Product and updates of the Licensed Product, including, but not limited to, rights in confidential and trade secret material, source code, object code, trademarks, service marks, patents, and copyrights, shall be and will remain the property of Licensor. Licensee shall have no right, title, or interest in such proprietary rights.

b. *Restrictions*. Licensee is prohibited from distributing, transferring possession of, or otherwise making available copies of the Licensed Product to any person not employed at the licensee's place of business. Licensee and Representative Authorized

Users are prohibited from making any modifications, adaptations, enhancements, changes, or derivative works of the Licensed Product, and Licensee shall advise all Authorized Users that they are prohibited from making any modifications, adaptations, enhancements, changes, or derivative works of the Licensed Product.

5. Confidentiality

a. *No Decompilation or Disassembly.* Licensor represents and Licensee hereby acknowledges that the object code constituting the Licensed Product and updates of the Licensed Product which is embodied on magnetic storage media contains confidential and trade secret material which is not readily susceptible to reverse compilation or reverse assembly. Licensee and Representative Authorized Users shall not attempt to decompile or disassemble the object code of the Licensed Product or updates. Licensee further agrees that it will use its best efforts to prevent decompilation and disassembly of the object code of the Licensed Product and updates by Authorized Users by advising Authorized Users of the provisions of this Subsection and by immediately reporting to Licensor and halting any reverse compilation or reverse assembly of the Licensed Product or updates by any Authorized User of which Licensee has actual knowledge.

6. Technical Support

Licensor, at its sole expense, shall provide Licensee with support of a technical nature with respect to all aspects of the Licensed Product and updates to the Licensed Product including their installation and use.

7. User Manuals

a. *Access to Manuals.* Licensee acknowledges that the user manual is an integral part of the software, which makes up the Licensed Product and is necessary for the proper use and application of the Licensed Product and updates to the Licensed Product.

b. *No Right to Copy Manual.* The license granted in Section 2 of this Agreement does not include any right to copy the user manual for use with the Licensed Product. Licensee acknowledges and agrees to use its best efforts to advise Authorized Users that any duplication of the manual is unauthorized by this Agreement, is prohibited by law,

and constitutes an infringement of Licensor's copyright. Violation of any provision in this Section shall be the basis for the immediate termination of this Agreement.

8. Limited Warranty and Disclaimer of Liability

a. *Results Not Warranted.* Licensor has no control over the conditions under which Licensee and Authorized Users use the Licensed Product and updates and does not and cannot warrant the results obtained by such use.

b. *Limited Warranty.* In addition to warranting that it has the right to grant the license contained in this Agreement, Licensor warrants that the magnetic media on which the Licensed Product or an update is recorded and any user manual leased under the terms of this Agreement are free from defects in material and workmanship under normal use. Licensor further warrants that the Licensed Product and any update of the Licensed Product will perform substantially in accordance with the specifications found in the user manual in effect as of the date of this Agreement. The warranties contained in this Section are made for a period of ninety (90) days from the date on which the Licensed Product or update is delivered to Licensee or from the date on which a user manual is leased by Licensee.

c. *Limitations on Warranty.* Licensor does not warrant that the functions contained in the Licensed Product or in any update will meet the requirements of Licensee or Authorized Users or that the operation of the Licensed Product or update will be uninterrupted or error free. The warranty does not cover any copy of the Licensed Product or update or any user manual, which has been altered or changed in any way by Licensee or any Authorized User. Licensor is not responsible for problems caused by changes in or modifications to the operating characteristics of any computer hardware or operating system for which the Licensed Product or an update is procured, nor is Licensor responsible for problems which occur as a result of the use of the Licensed Product in conjunction with software or with hardware which is incompatible with the operating system for which the Licensed Product is being procured.

d. *Exclusion of Implied Warranties.* ANY IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ARE EXPRESSLY EXCLUDED.

a. *Exclusion of Any Other Warranties.* The warranties contained in Subsection b of this Section are made in lieu of all other express warranties, whether oral or written. Only

an authorized officer of Licensor may make modifications to this warranty or additional warranties binding on Licensor, and such modifications or additional warranties must be in writing. Accordingly, additional statements such as those made in advertising or presentations, whether oral or written, do not constitute warranties by Licensor and should not be relied upon as such.

11. Limitation of Remedies

a. *Replacement Sole Remedy.* Subject to Section 22 of this Agreement, Licensor's entire liability and Licensee's exclusive remedy shall be the replacement by Licensor of any magnetic media or user manual not meeting Licensor's "Limited Warranty." In addition, while in no sense warranting that the operation of the Licensed Product will be uninterrupted or error free, Licensor will, as provided in Section 7 of this Agreement, assist the Licensee in the installation and maintenance of the Licensed Product and will supply the Key Person with corrected versions of the Licensed Product through updates.

b. *Damages Limitation.* LICENSOR DISCLAIMS ANY AND ALL LIABILITY FOR SPECIAL,

INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOSS OF PROFIT) ARISING OUT OF THIS AGREEMENT OR WITH RESPECT TO THE INSTALLATION, USE, OPERATION, OR SUPPORT OF THE LICENSED PRODUCT OR ANY UPDATE OF THE LICENSED PRODUCT EVEN IF LICENSOR HAS BEEN APPRISED OF THE POSSIBILITY OF SUCH DAMAGES.

c. *Limitation of Any Recover,'* Subject to Section 22 of this Agreement, Licensee specifically agrees that any liability on the part of Licensor arising from breach of warranty, breach of contract negligence, strict liability in tort, or any other legal theory shall not exceed amounts paid by Licensee in fees for the use and maintenance of the Licensed Product.

12. Relationship of the Parties

For purposes of this Agreement, Licensee is not an agent of Licensor, and Licensee has no express or implied authority to act on behalf of or make any representations whatsoever on behalf of Licensor. Licensor has no right to control any activities of Licensee outside the terms of this Agreement.

13. Updates

Updates include enhancements and corrections of and modifications and additions to the Licensed Product. Updates also include later versions of the Licensed Product. Use of updates with or in place of the Licensed Product shall be fully governed by and subject to the terms of this Agreement relating to the reproduction and use of the Licensed Product. Any portion of the Licensed Product replaced by an update shall be destroyed.

14. Supplements

From time to time, Licensor will make available computer programs which are compatible with the

Licensed Product and which supplement the Licensed Product. SUPPLEMENTS ARE NOT

LICENSED UNDER THE TERMS OF THIS AGREEMENT.

15. Indemnity

Licensor, at its own expense, will defend any action brought against Licensee to the extent that it is based or, a claim that the Licensed Product or any update of the Licensed Product used within the scope of this Agreement infringes any patent, copyright, license, trade secret, or other proprietary right, provided that Licensor is immediately notified in writing of such a claim. Licensor shall have the right to control the defense of all such claims, lawsuits, and other proceedings. In no event shall Licensee settle any such claim, lawsuit, or proceeding without Licensor's prior written approval. Licensor shall have no liability for any claim under this Section if a claim for patent, copyright, license, or trade secret infringement is based on the use of a superseded or altered version of the Licensed Product if such infringement would have been avoided by use of the latest unaltered version of the Licensed Product available as an update.

16. Arbitration

Except for the right of either party to apply to a court of competent jurisdiction for a temporary restraining order, a preliminary injunction, or other equitable relief to preserve

the status quo or prevent irreparable harm, any controversy or claim arising out of or relating to this Agreement or to its breach shall be settled by an arbitration administered by the American Arbitration Association and pursuant to its rules, and judgment upon the award rendered in such arbitration may be entered in any court of competent jurisdiction.

17. General

a. *Complete Agreement; Amendment.* Each party acknowledges that it has read this Agreement and any exhibit, understands them, and agrees to be bound by their terms, and further agrees that they are the complete and exclusive statement of the agreement between the parties which supersedes and merges all prior proposals, understandings, and all other agreements, oral and written, between the parties relating to this Agreement. This Agreement may not be modified or altered except by written instrument duly executed by both parties.

b. *Purchase Order.* In the event of any conflict between the terms and conditions of this Agreement and the terms and conditions of any purchase order, the terms and conditions of this Agreement shall control.

c. *Governing Laws.* The laws of the State of Texas shall govern this Agreement and performance under this Agreement

d. *Limitations Period.* No action, regardless of form, arising out of this Agreement may be brought by Licensee more than two (2) years after the cause of action has arisen.

e. *Severability.* If any provision of this Agreement is invalid under any applicable statute or rule of law, it is to that extent to be deemed omitted. The remainder of the Agreement shall be valid and enforceable to the maximum extent possible.

f. *Assignment.* Licensee may not assign or sublicense, without the prior written consent of Licensor, its rights, duties, or obligations under this Agreement to any person or entity, in whole or in part.

g. *Assumption by Successor to Licensor.* In the event of the acquisition of Licensor's business, software, or both by a third party, Licensor agrees to make such an acquisition subject to the assumption of the terms of this Agreement by the third party.

h *Cessation of Business.* Should Licensor cease doing business for reasons other than the acquisition of the business or software by a third party, the license granted in Section 2 of this Agreement shall become a perpetual, nonexclusive, nontransferable license. The provisions of Sections 5 and 6 of this Agreement shall apply fully to such a license.

j. *Waiver*. The waiver or failure of Licensor to exercise in any respect any right provided for in this Agreement shall not be deemed a waiver of any further right under this Agreement.

k. *Headings.* The headings appearing at the beginning of the several sections contained in this Agreement have been inserted for identification and reference purposes only and shall not be used in the construction and interpretation of this Agreement.

Technical Support

Please note that currently supported platforms for the software are Windows NT, Windows 2000, and Windows XP. Other older platforms (Windows 98, Windows 95) might work but are not supported.

The latest release is available from: http://www.robohandsla.com/software/setup.zip. To obtain an earlier release, please contact us.

If you need any technical support for installation or using the SLA OS, please contact DE-STA-CO Automation at the following information.

DE-STA-CO Automation
3305 Wiley Post
Carrollton, TX 75006
tech-sla@robohand.com
http://www.robohandsla.com/
(800)-259-9890 / 972-726-7300

Quick Start

Quick Introduction to Configuration of SLA OS

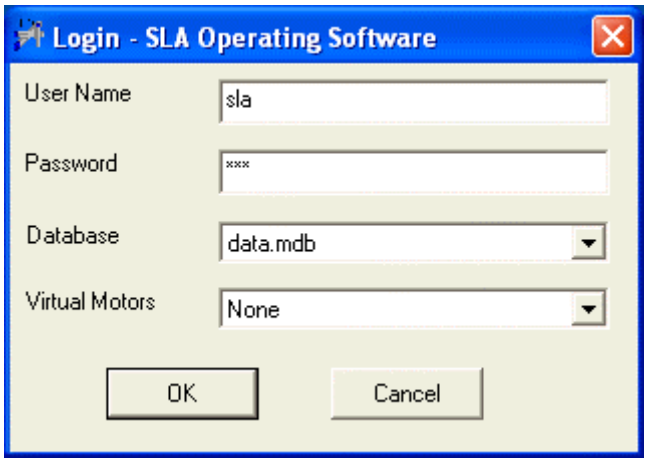
This section is designed for a quick introduction to the SLA OS by showing how easy it is to write useful programs to get the results you want. It shows examples of three different types of programs:

1. **Multi Motor Program (MMP)** - can handle most applications with its inbuilt library of highly sophisticated commands. It runs from the (optional) SLA Control Unit.
2. **Smart Motor Program (SMP)** - runs in the motors (doesn't require SLA Control Unit) and can handle many of the less complex tasks.
3. **Binary Coded Decimal Program (BCD)** - type of simplified SMP program where PLC based controls can be used to send BCD numbers to the motors to go to pre programmed locations.

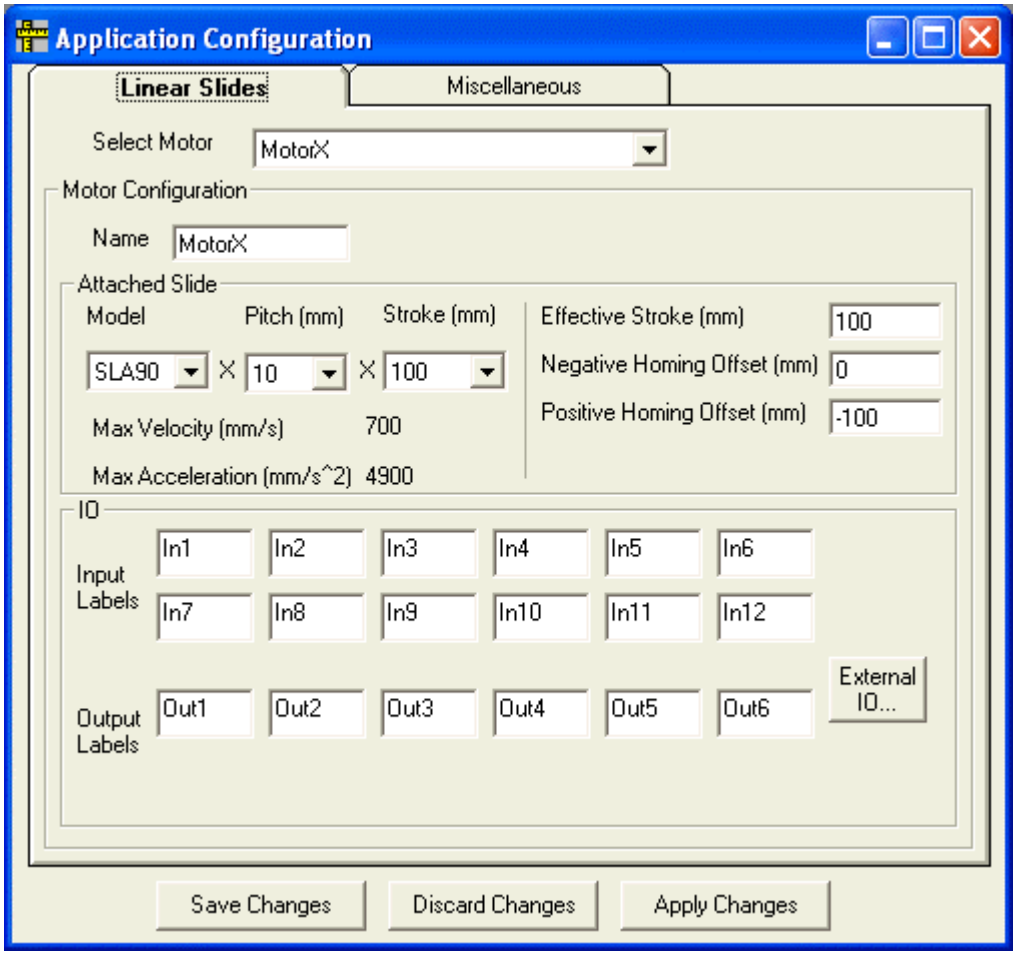
Before a program can be written the SLA OS software must be configured to match the settings of the slides configuration.

Quick Configuration

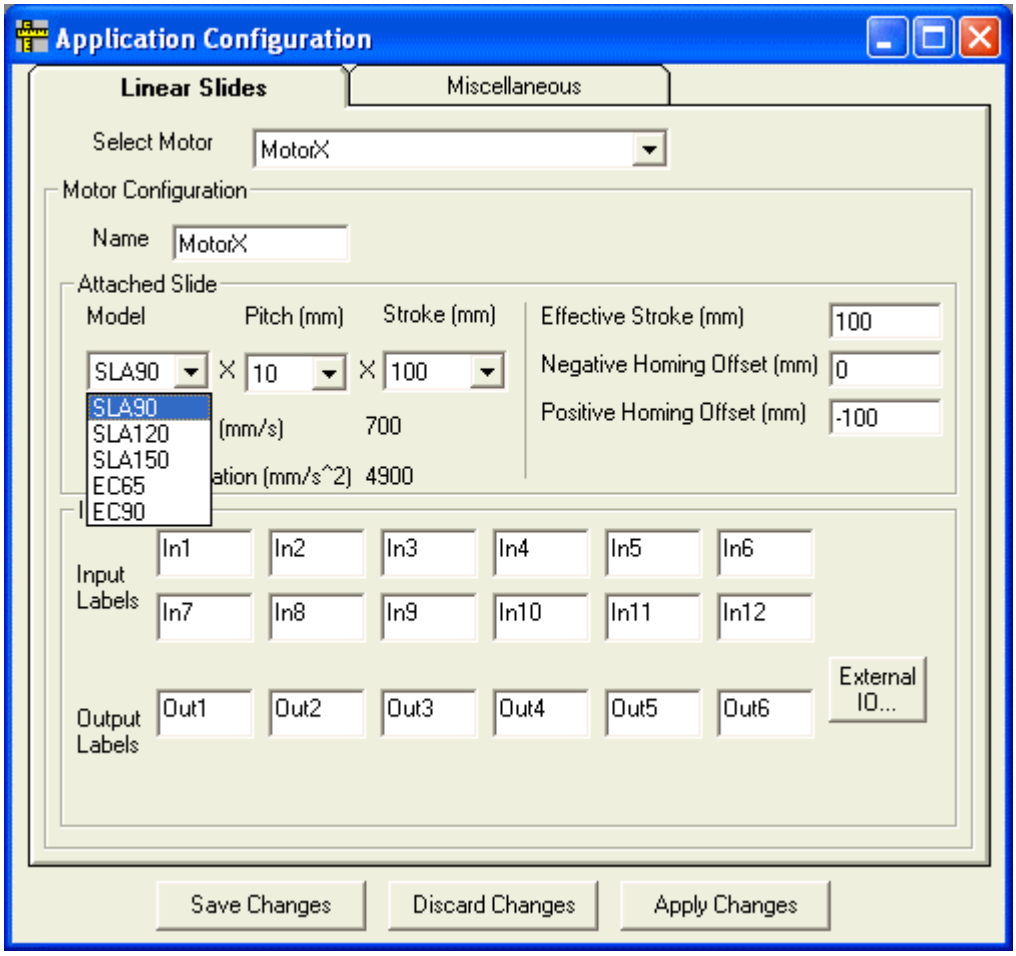
Once the SLA OS software is installed (it already comes pre-installed if you also ordered the SLA Control Unit), login with the user and password both as "sla" (in lowercase).



Click on the Configuration icon in the toolbar to open the following configuration screen.



Select the correct slide models and stroke lengths for each of the motors by clicking on the appropriate drop down combobox and saving the settings.



Once the system is configured, the following three sections show how to quickly write programs using the three mentioned program types.

1. **Multi Motor Program (MMP) Quick Start**
2. **Smart Motor Program (SMP) Quick Start**
3. **Binary Coded Decimal Program (BCD) Quick Start**

Later in the manual many of the advanced features will be discussed in more details to fully harness the power of SLA OS.

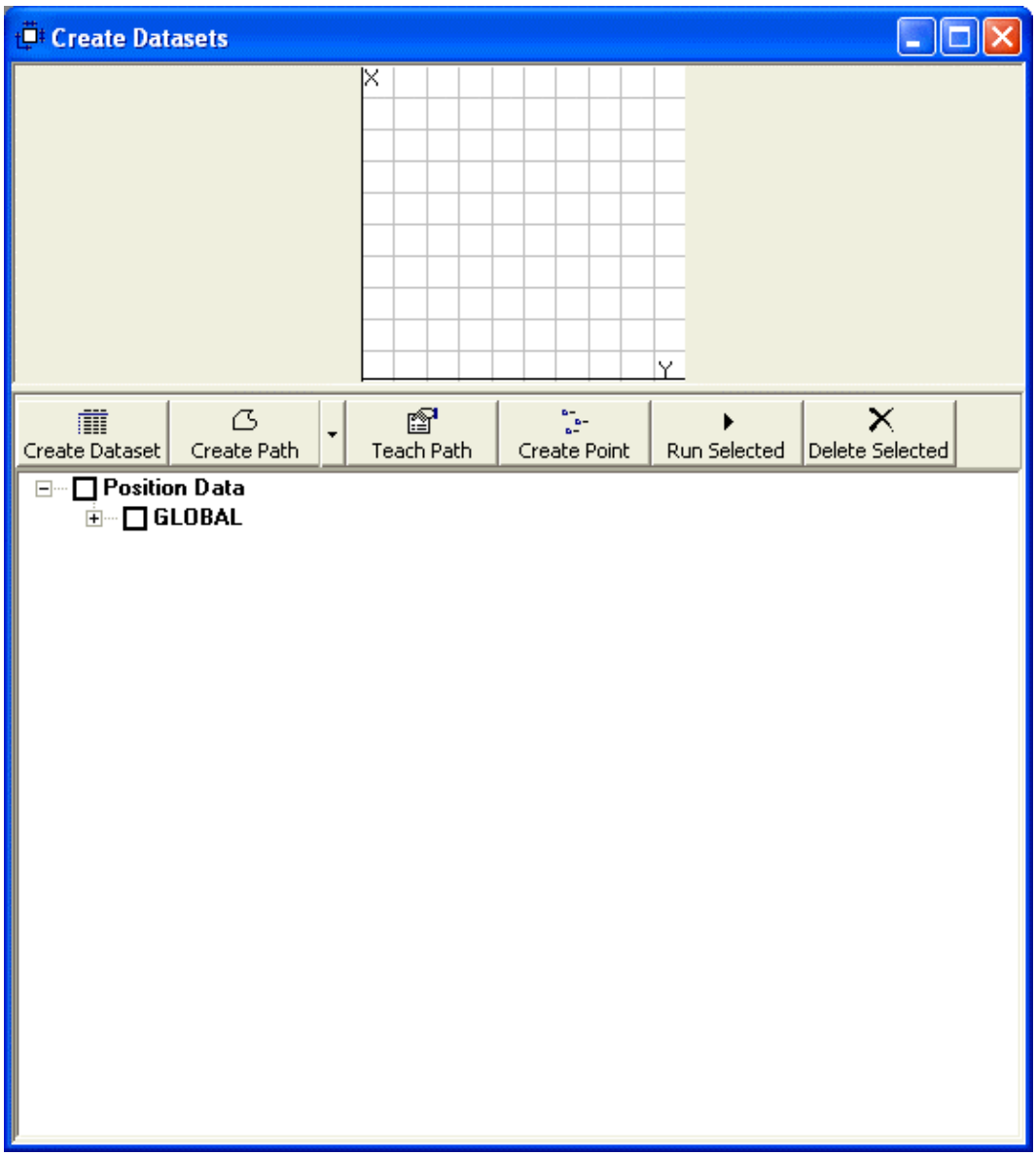
Quick Introduction to Multi Motor Programming with SLA OS

This section is designed for a quick introduction to Multi Motor Programming with SLA OS by writing a simple Multi Motor Program (MMP) for dispensing along a square shape of 60 mm side with rounded corners of radius 5 mm. The three main steps are

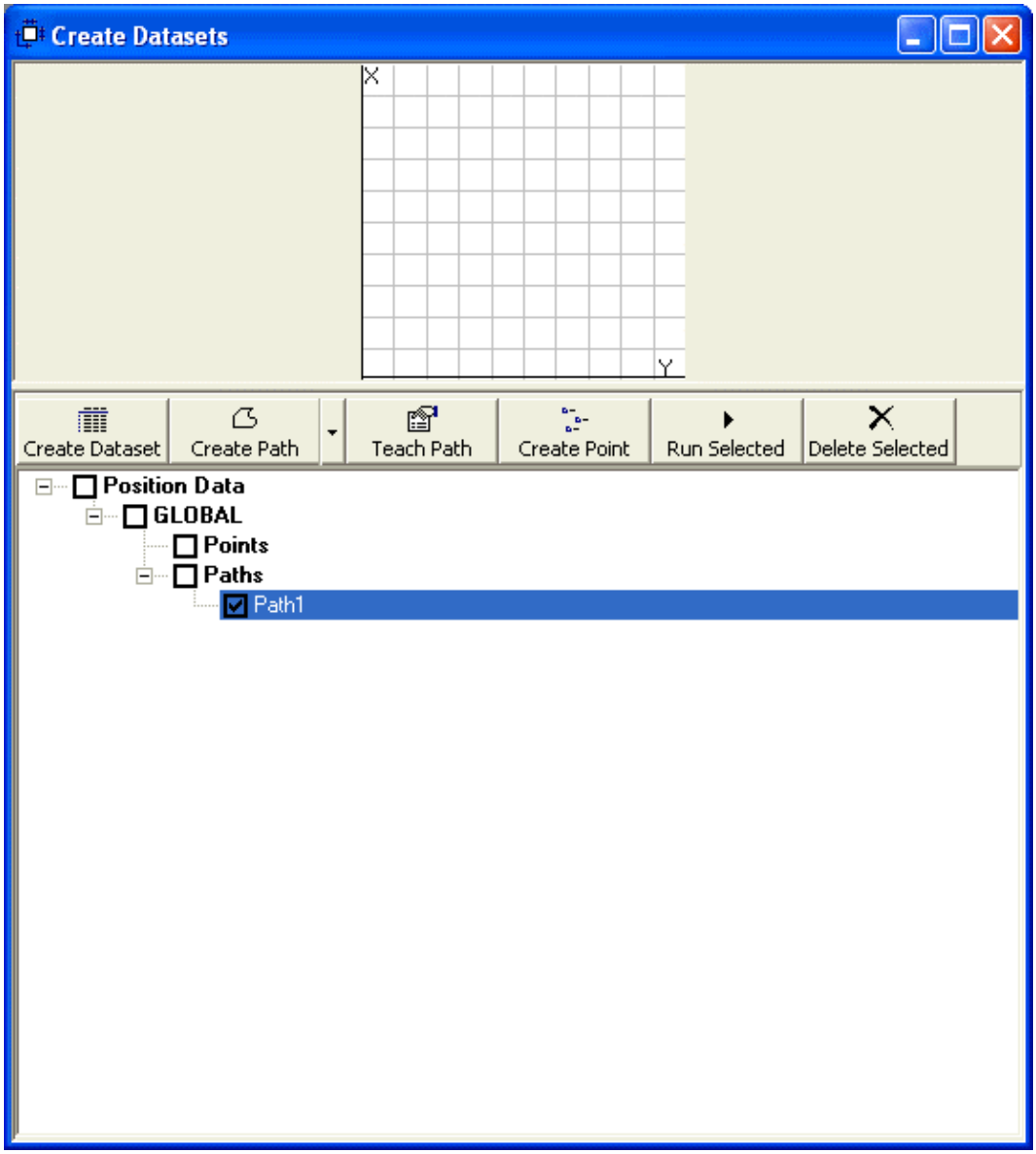
1. Position Data Creation
2. Programming
3. Results

Quick Position Data Creation

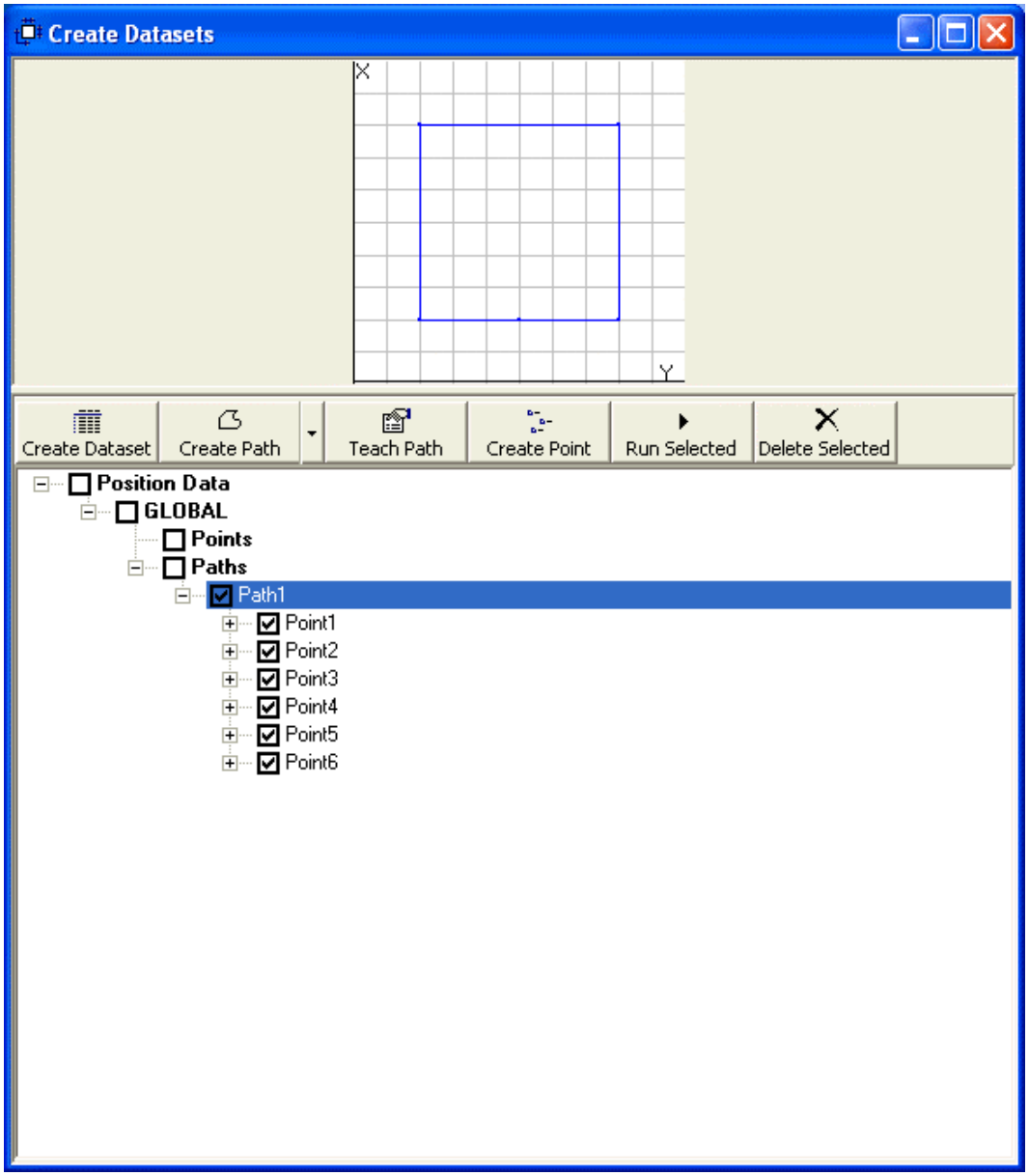
Click on the Datasets icon in the toolbar to open 'Create Datasets' screen.



Click on the 'Create Path' icon in the screen to create a new path for dispensing.

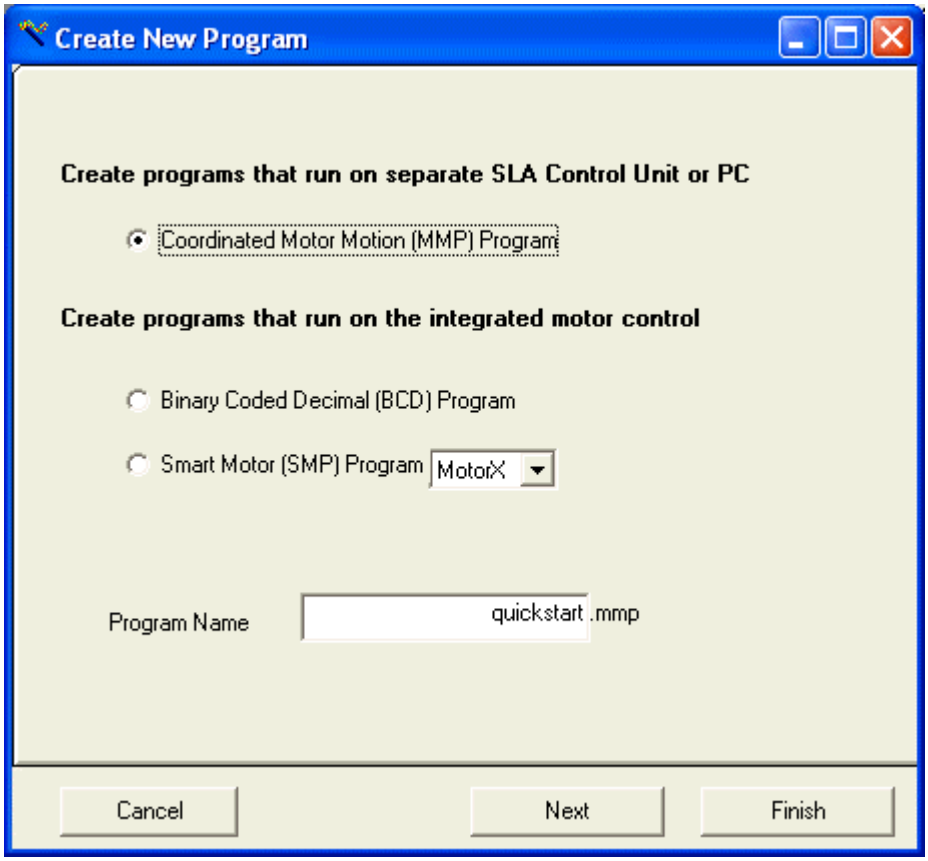


With the mouse, create the desired path as shown below. The six point square includes starting from the center of a side and selecting all four corners and back to the original center point to create six points as shown below.

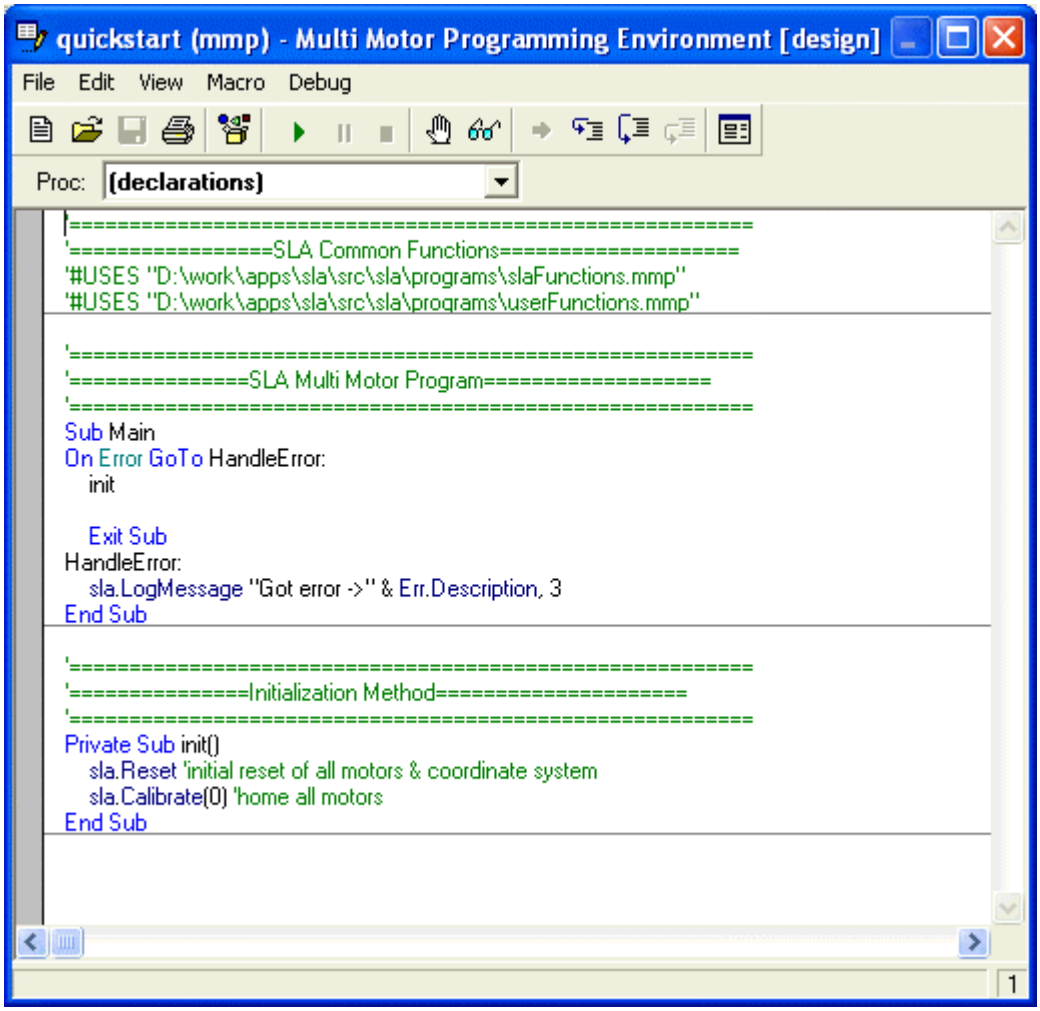


Quick Programming

Open a 'New Program...' dialog by clicking on File menu or pressing Ctrl+N (Control and N together). By default, the MMP program is selected. Type the name of the new program in 'Program Name' textbox and click 'Create Program' button.

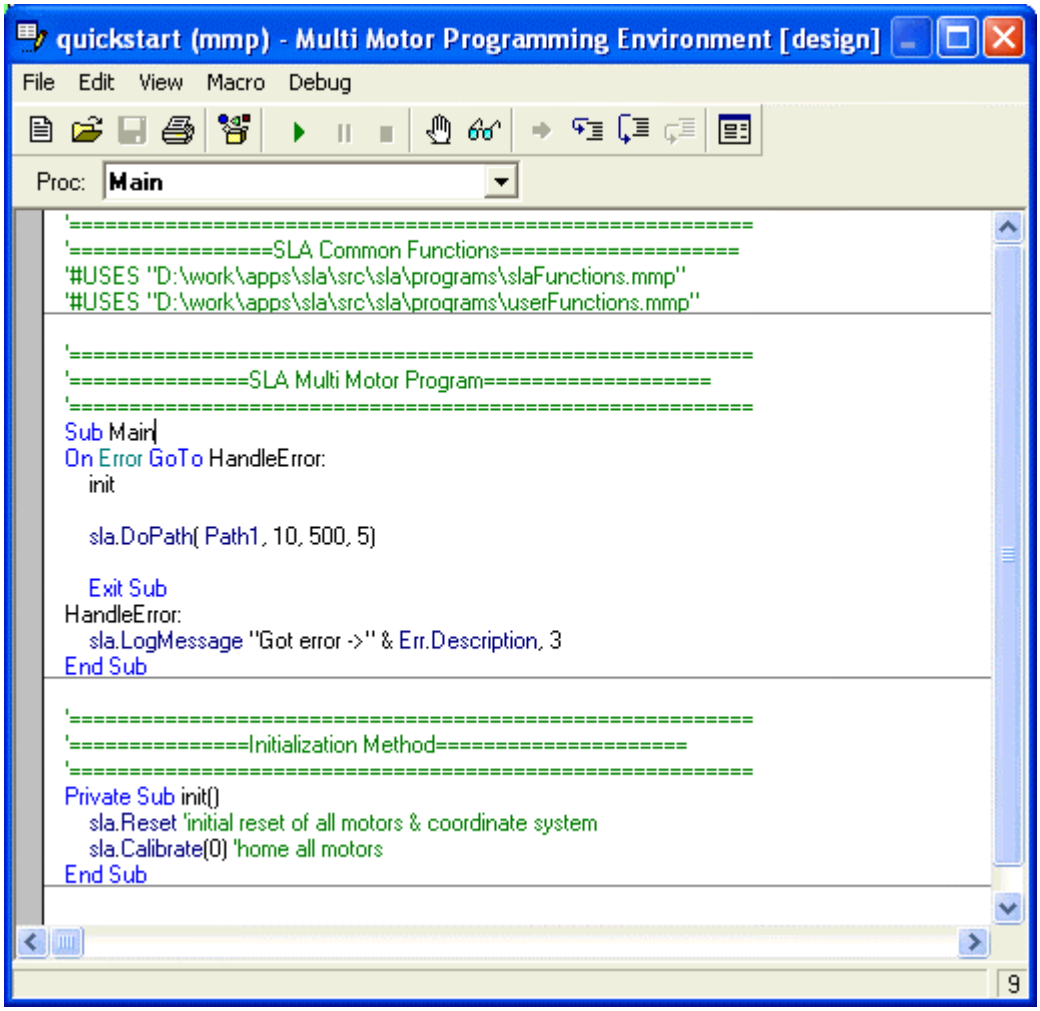


This will bring up the MMP Programming Environment with the template of the program already created as shown below.



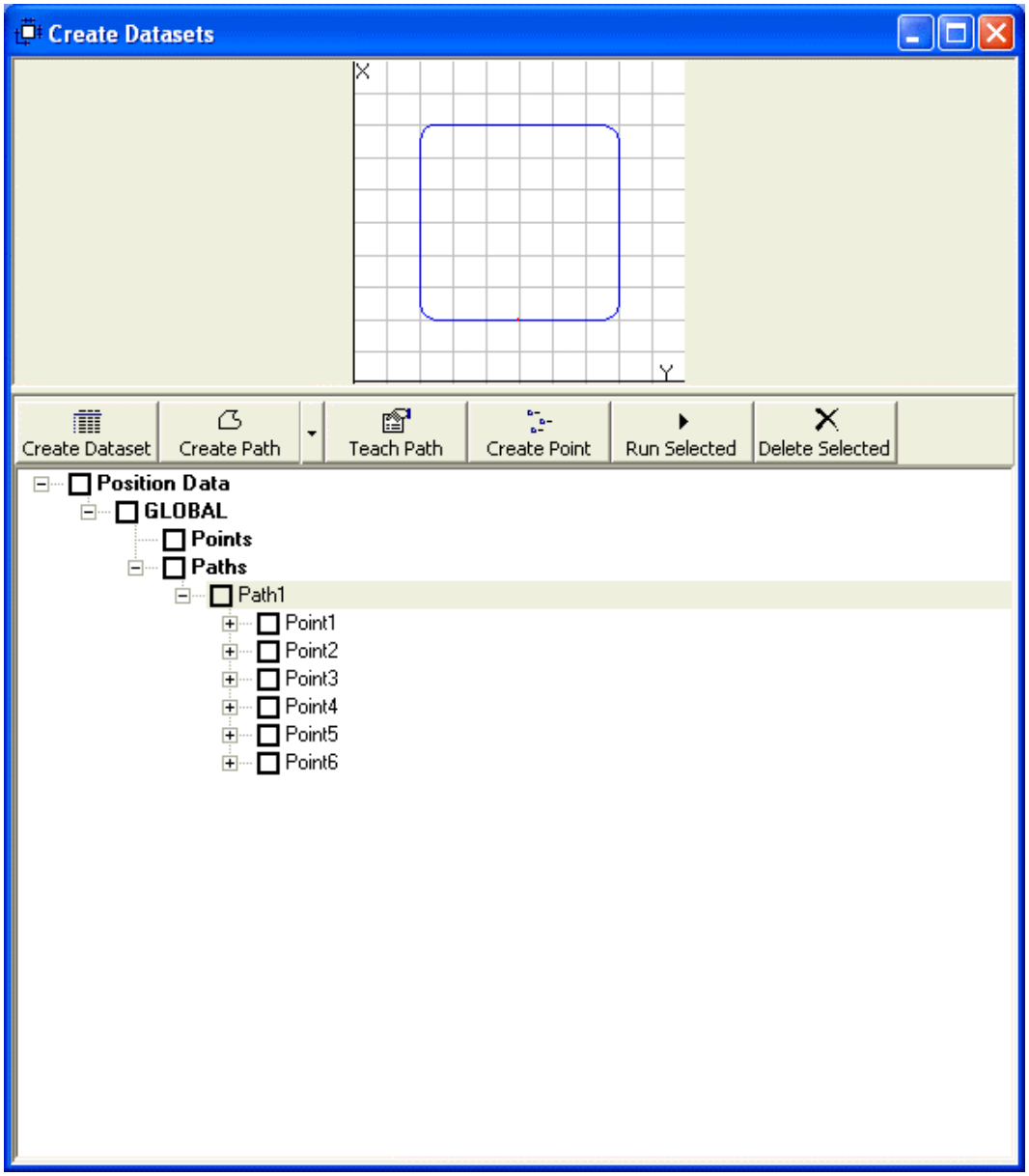
Just after the 'init' call to calibrate the system, add the following line to make the slide follow the square path (Path1) just created. Note that as soon as you start typing 'sla.' the autocomplete feature of the environment displays all the possible completion. At this point it is a matter of selecting from the options and filling in the arguments. The following command will make slide follow the Path1 with 10 mm/sec. speed and acceleration of 500 mm/sec.^2. Note, how easy it is to add the rounding of 5 mm for the corners.

```
sla.DoPath( Path1, 10, 500, 5)
```



Quick Results

Select the 'Monitor All' and 'Visual Trace' checkboxes on the main window to follow the movement of the slides on the Dataset screen. Click the Start button on the MMP Programming Environment to run the program to see the following successful result.


Create Datasets
X
Y
Create Dataset
Create Path
Teach Path
Create Point
Run Selected
Delete Selected
Position Data
GLOBAL
Points
Paths
Path1
Point1
Point2
Point3
Point4
Point5
Point6

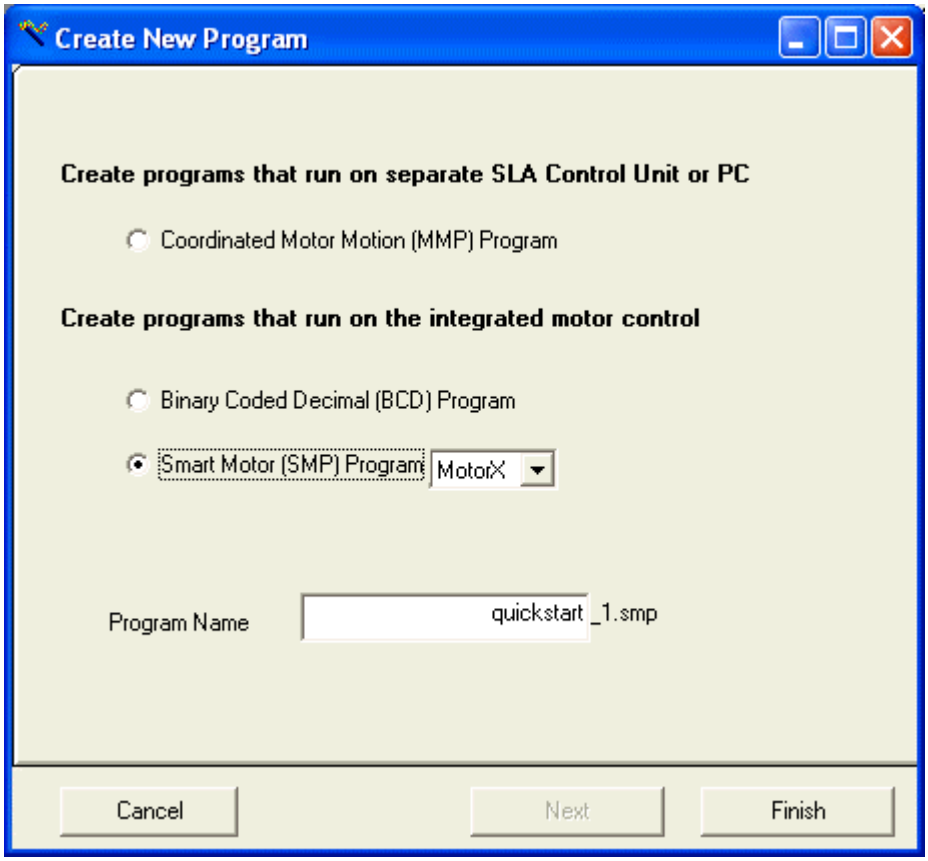
Quick Introduction to Smart Motor Programming with SLA OS

This section is designed for a quick introduction to Smart Motor Programming with SLA OS by writing a simple Smart Motor Program (SMP) for creating a constant velocity motion of for MotorX in positive direction. The two main steps are

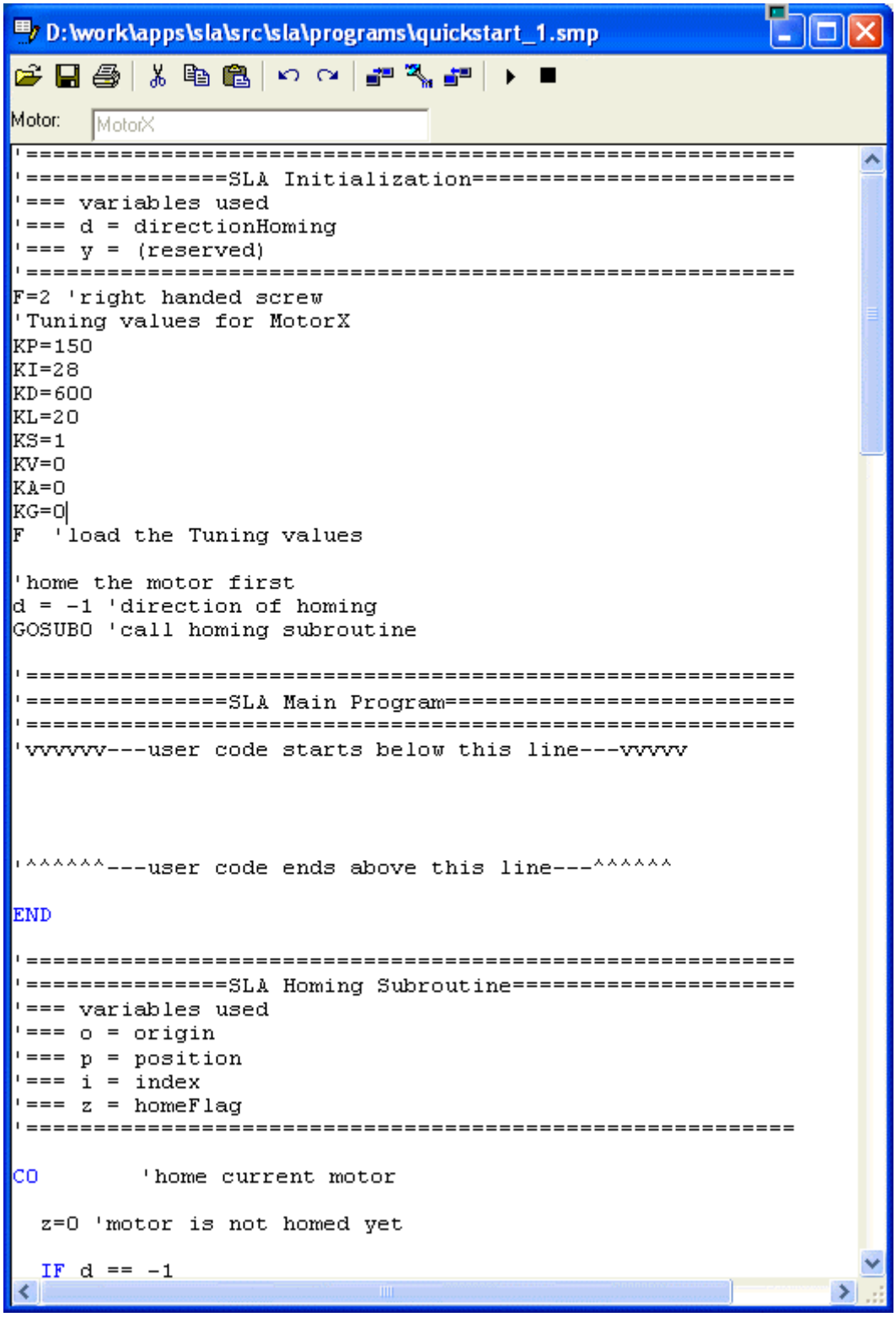
1. Programming
2. Results

Quick Programming

Open a 'New Program...' dialog by clicking on File menu or pressing Ctrl+N (Control and N together). Select the SMP program type and enter the name of the new program in 'Program Name' textbox. Note that by default, MotorX is selected. Click 'Create Program' button.



This will bring up the SMP Programming Environment with the template of the program already created as shown below which includes the instructions for downloading tuning parameters and homing the motor at the start of the program.



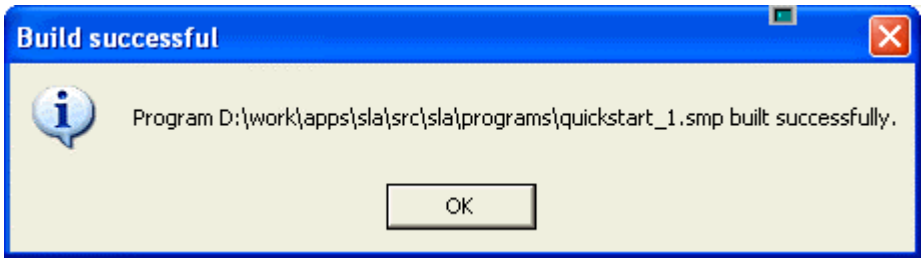
Between the section indicated for user code, add the following lines to make the slide move in the positive X direction with 1000000 Smart Motor Unit speed and 100 Smart Motor Unit acceleration. The command MV defines the velocity mode for motion and the command G starts the motion.

```
MV
A = 100
V = 1000000
G
```

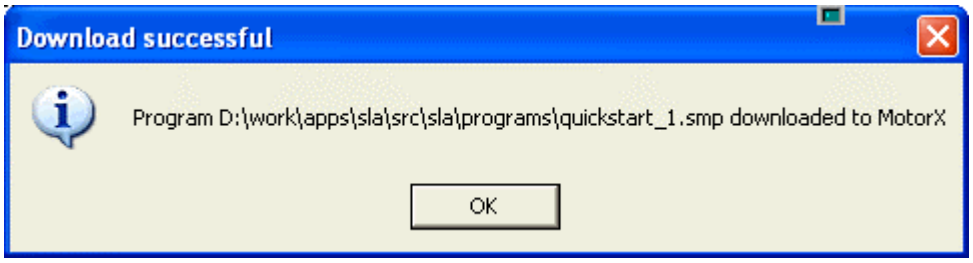

D:\work\apps\sla\src\sla\programs\quickstart_1.smp*
Motor:
MotorX
'==
'===============SLA Initialization=========================
'=== variables used
'=== d = directionHoming
'=== y = (reserved)
'==
F=2 'right handed screw
'Tuning values for MotorX
KP=150
KI=28
KD=600
KL=20
KS=1
KV=0
KA=0
KG=0
F 'load the Tuning values
'home the motor first
d = -1 'direction of homing
GOSUB0 'call homing subroutine
'==
'===============SLA Main Program============================
'==
'vvvvvv---user code starts below this line---vvvvv
MV
A = 100
V = 1000000
G
'^^^^^^---user code ends above this line---^^^^^^
END
'==
'===============SLA Homing Subroutine======================
'=== variables used
'=== o = origin
'=== p = position
'=== i = index
'=== z = homeFlag
'==
CO   'home current motor
  z=0 'motor is not homed yet

Quick Results

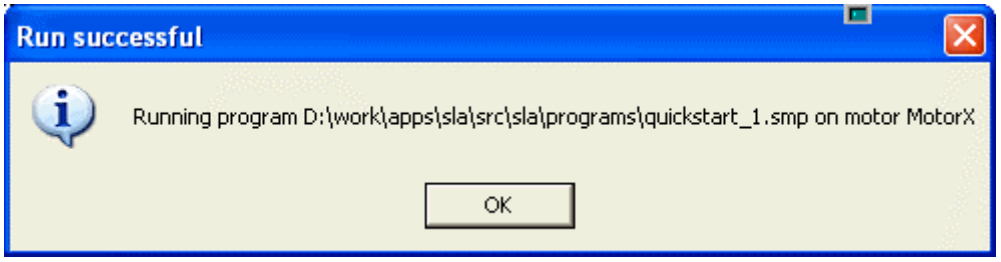
Before you can run the program, first build the program by clicking on the 'Build the Smart Motor Program' button in the toolbar. You should see the following confirmation to make sure that the program compiles successfully.



Next step is to download the compiled program to motor. Click on the 'Transfer Program from PC to Motor' button in the toolbar. You should see the following confirmation dialog box.



Now, to run the program, click on the 'Run' button. This will result in the downloaded programs running in the destination MotorX. By default, the generated programs run the homing routine initially. This behavior can be changed by modifying the program and rebuilding and downloading again.



The slide will move in the positive X direction and come to stop at the limit switch at the end of the slide.

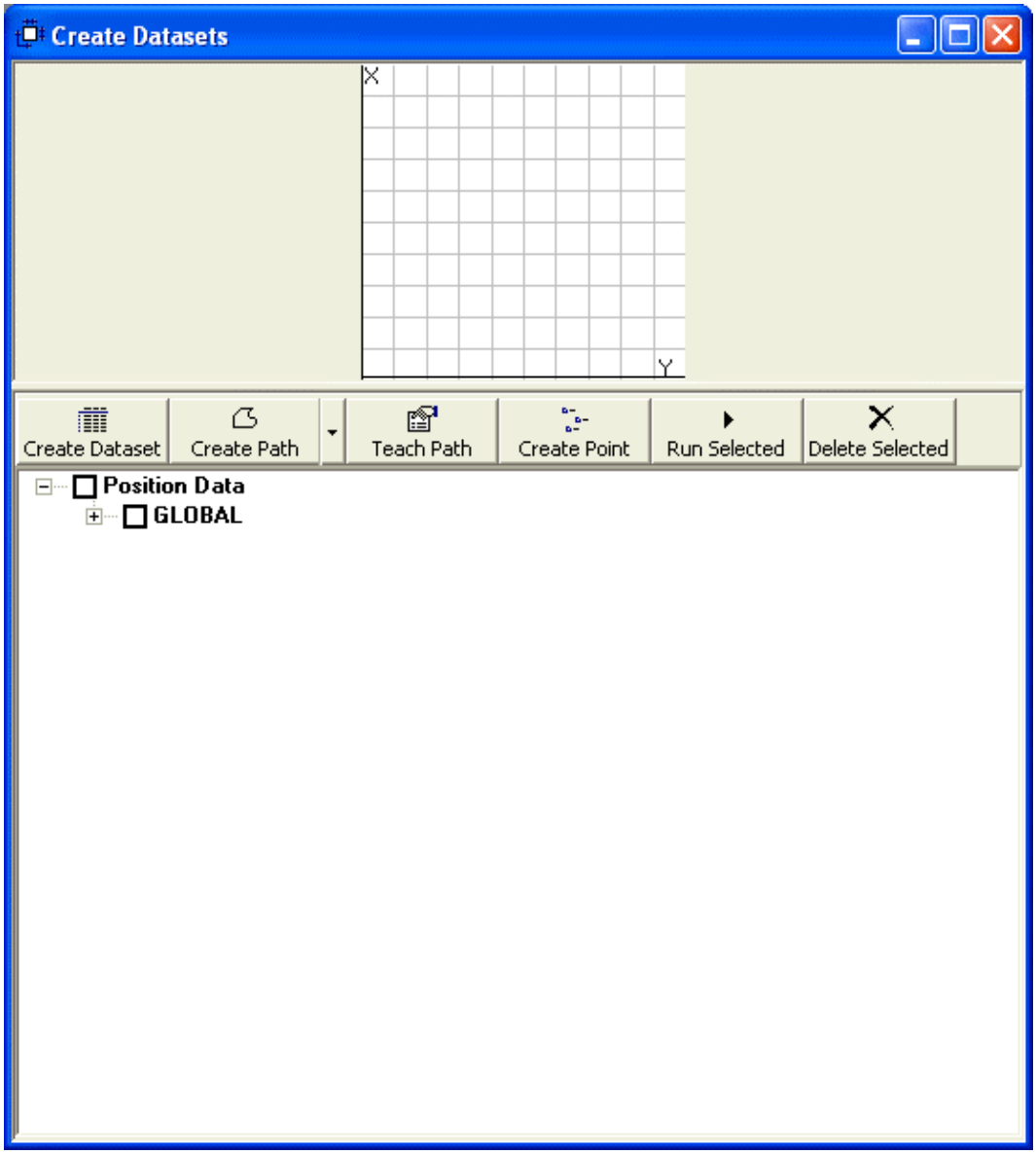
Quick Introduction to Binary Coded Decimal Programming with SLA OS

This section is designed for a quick introduction to Binary Coded Decimal Programming with SLA OS by writing a simple Binary Coded Decimal Program (BCD) for generating series of positions to move The three main steps are

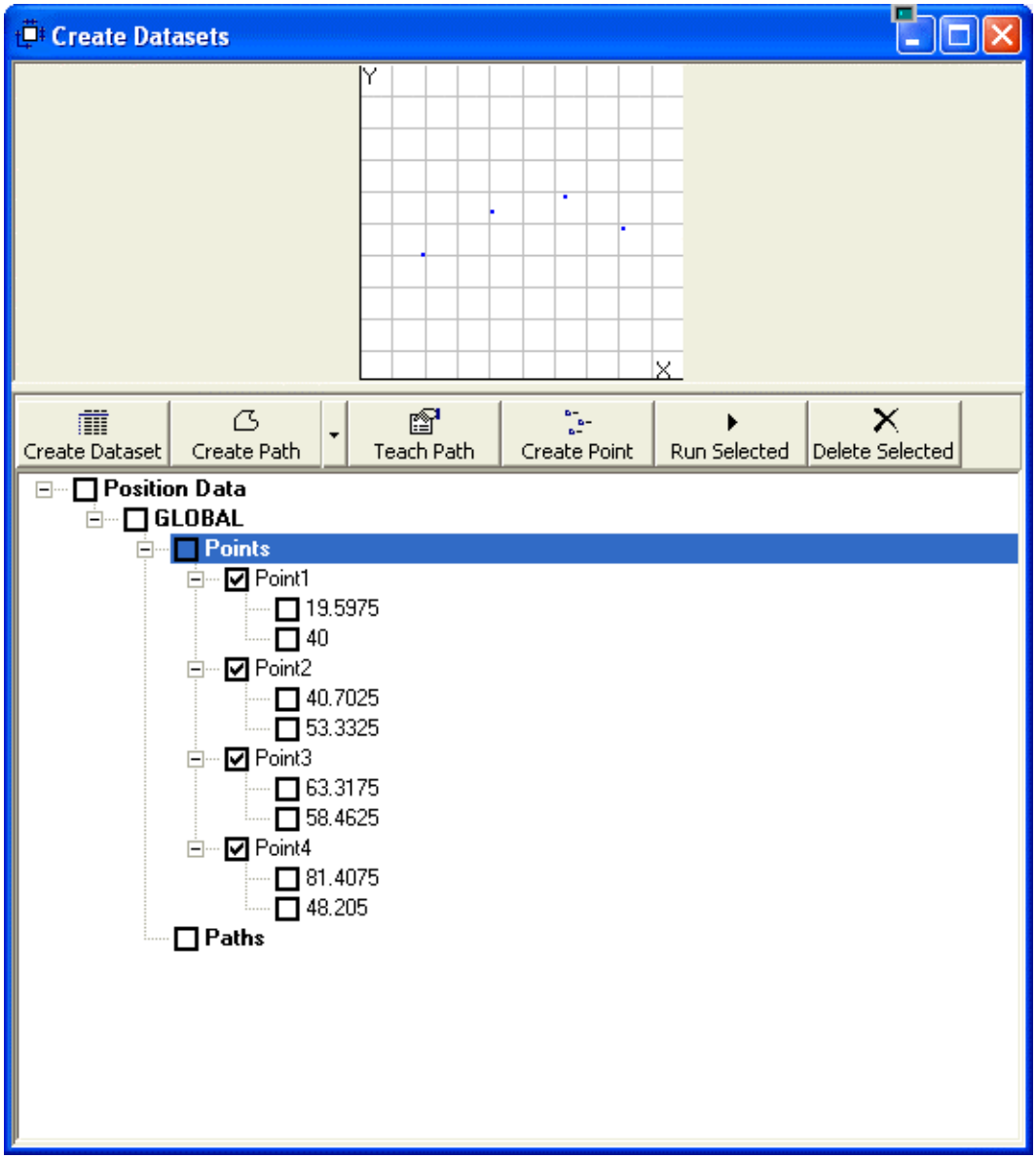
1. Position Data Creation
2. Programming
3. Results

Quick Position Data Creation

Click on the Datasets icon in the toolbar to open 'Create Datasets' screen.

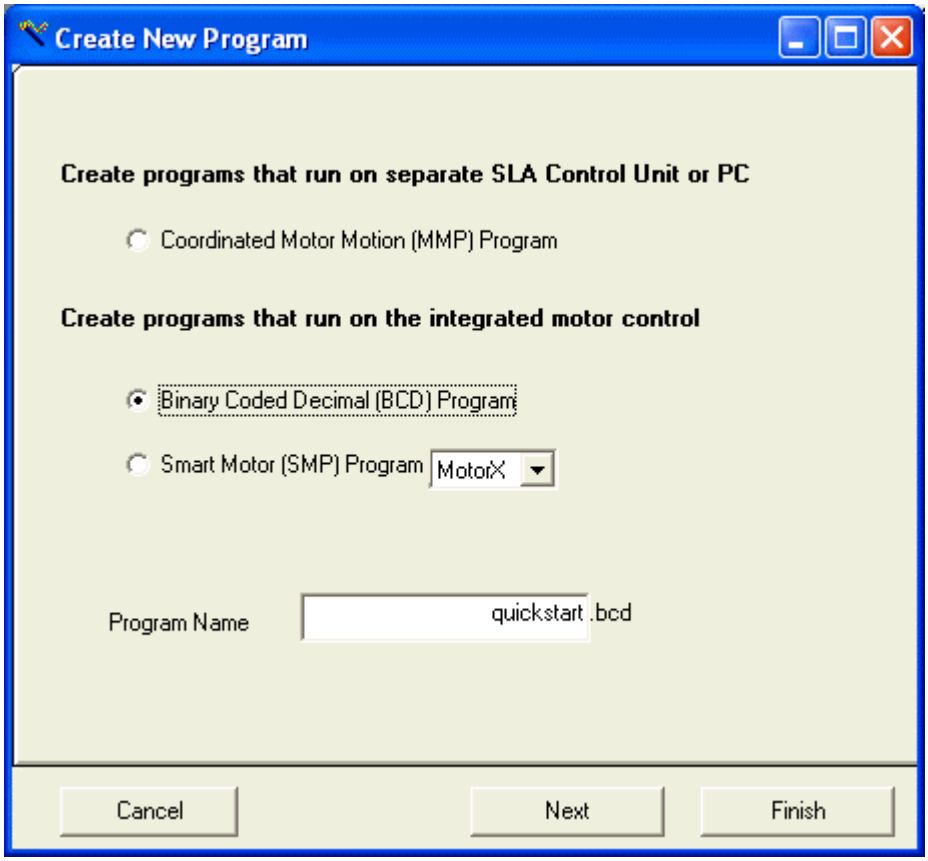


With the mouse, click on the screen to create the positions corresponding to the BCD input. With each click, a new point is added to the Points node as shown below. Since this is just for the test purpose, the exact locations of the points don't matter.

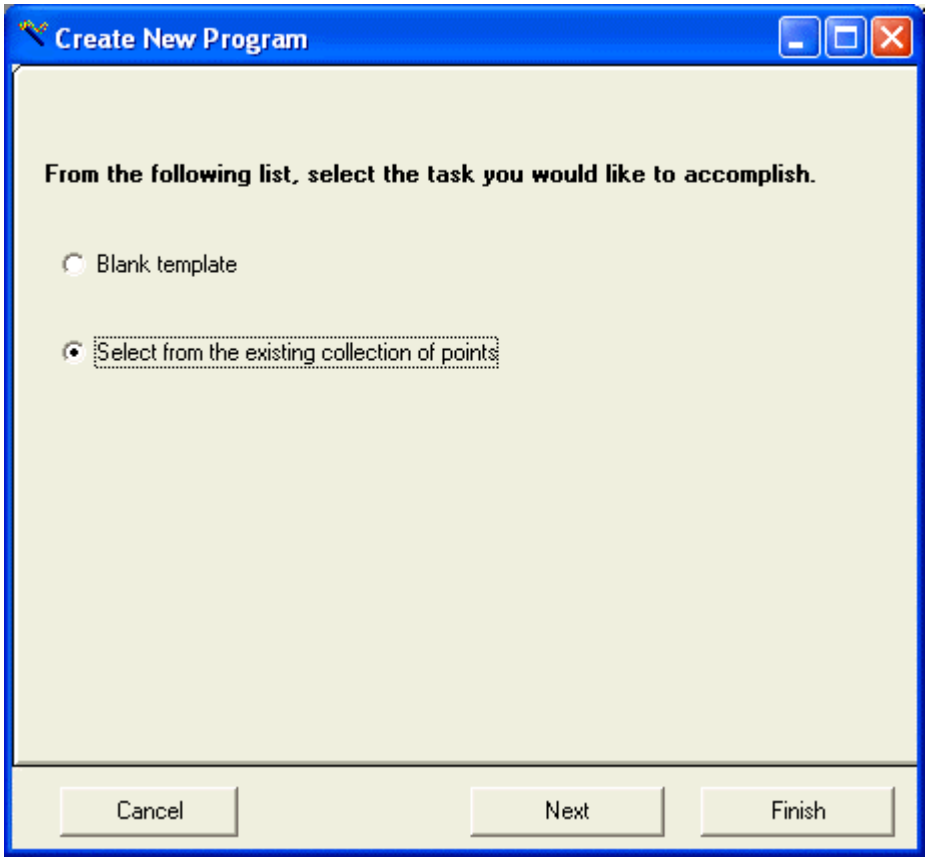


Quick Programming

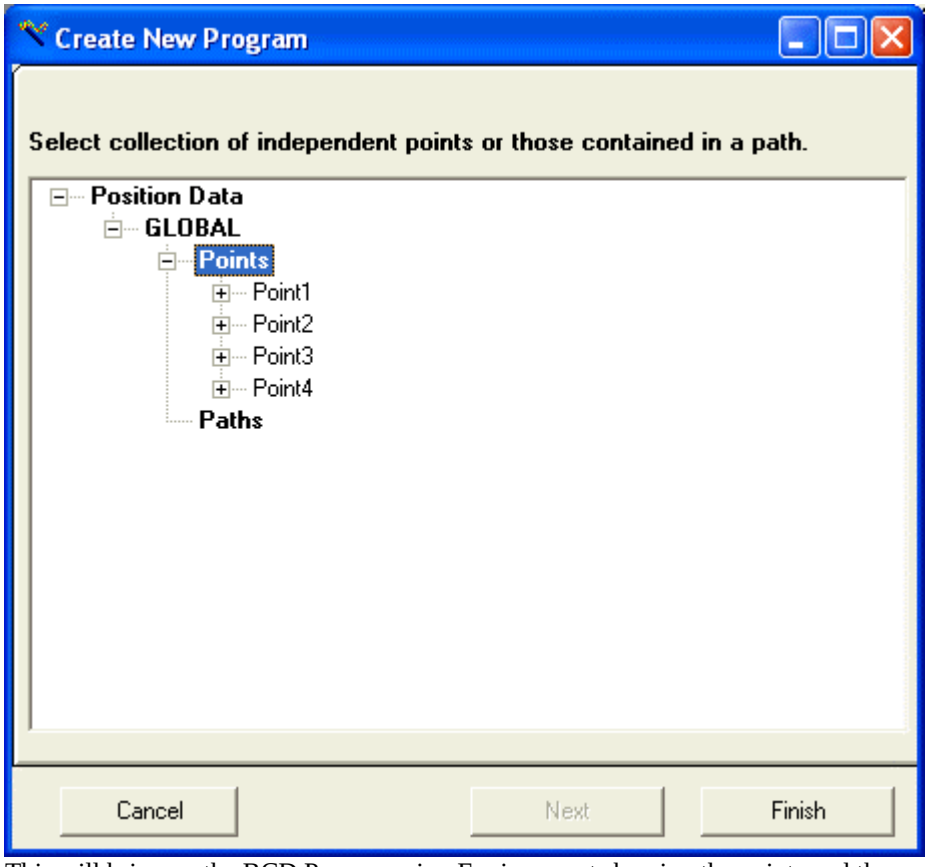
Open a 'New Program...' dialog by clicking on File menu or pressing Ctrl+N (Control and N together). Select the BCD program type and enter the name of the new program in 'Program Name' textbox. Click 'Next' button.



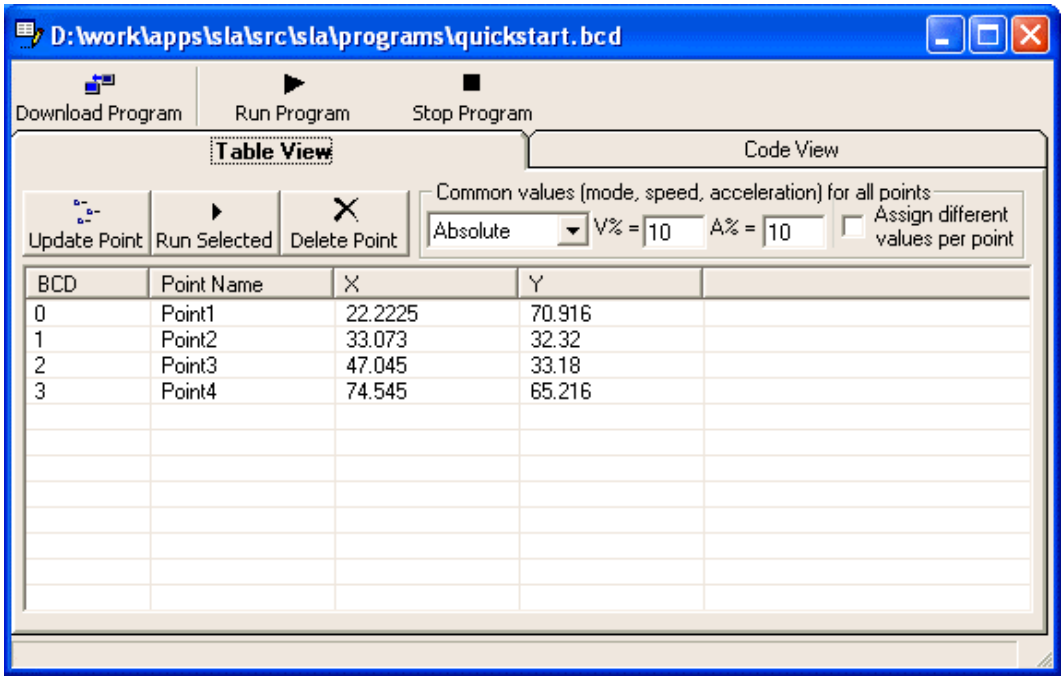
In the task selection window, you can select different ways to create a BCD program. Select the second option for creating a BCD program from existing data.



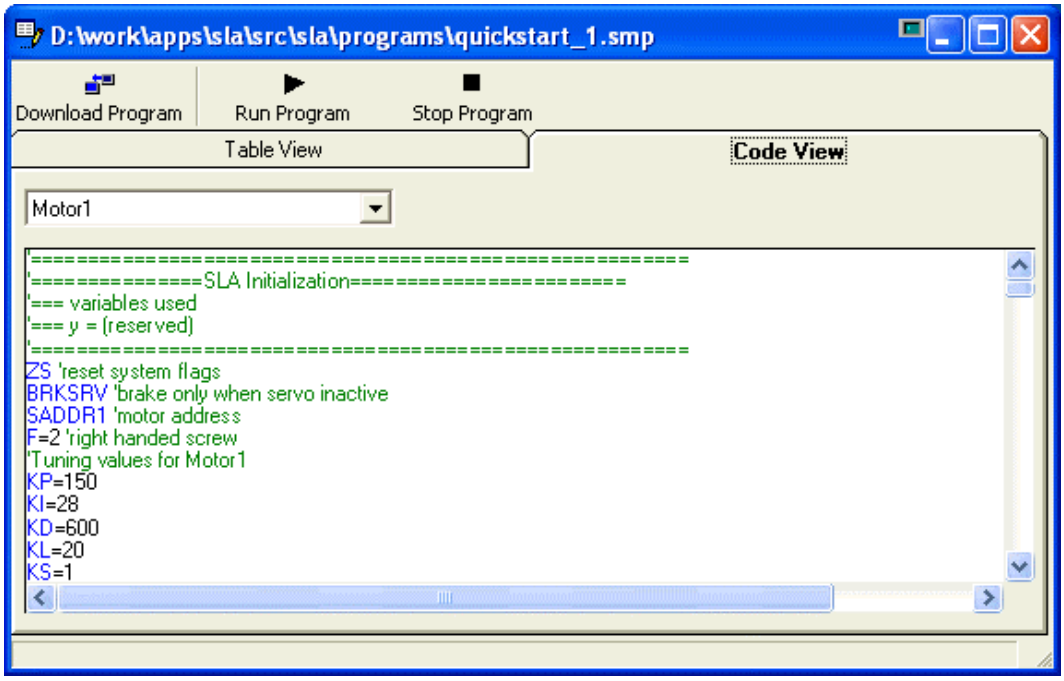
This will bring up a window with the position data tree that was earlier created in the Datasets window. Select the collection of points (either a path or points) to create BCD program from as shown below.



This will bring up the BCD Programming Environment showing the points and the corresponding BCD values.

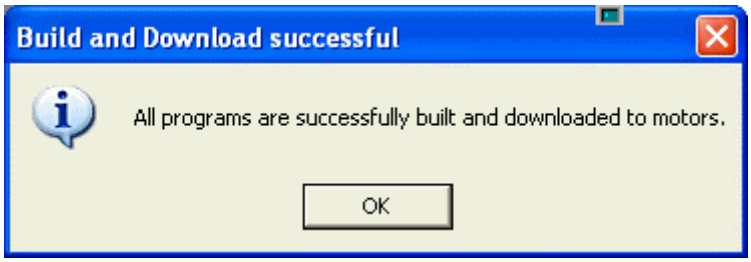


Click on the 'Code View' tab to see the SLA OS generated code which also includes code for homing to match the existing configuration information. Note that the displayed code depends on the type of slide connected to the axis. The following display corresponds to SLA.

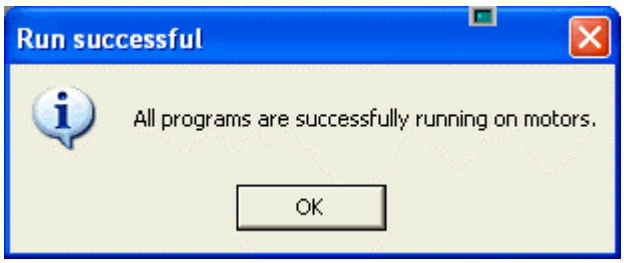


Quick Results

Since all the code is already generated by the SLA OS software, there is no additional programming involved. Before you can run the program, compile and download the programs to all the motors by clicking on the 'Build and Download' button on the top. You should see the following confirmation to make sure that everything is successful.



Now, to run the program, click on the 'Run' button. This will result in the downloaded programs running in all the motors. By default, the generated programs run the homing routine initially. This behavior can be changed using the SMP Programming Environment since BCD programs are basically SLA OS generated SMP programs.



Once the programs are running, the motion can be achieved by feeding the correct input values using either a test BCD input box or PLC.

Introduction

Installing SLA OS Software

To install the software:

1. Close all programs including any copy of SLA OS Software.
2. If you have a previous copy of the software installed, uninstall it before installing the new version. (note: Older versions (before v1.2) require backing up of database located in "C:\Program Files\Robohand\SLA\data\data.mdb" in a safe location.) To uninstall the previous version, open Control Panel, click on 'Add or Remove Programs', click on 'SLA Operating Software' and follow the instructions for removing the software.
3. Double click on the installation setup.exe file. The installation wizard will lead through the steps for installation.
4. If you had to backup the database in earlier step, copy the database back in the original location.
5. Once the software is installed, it will create shortcuts on the Desktop as well as create additional menu items in the program menu accessible from standard Windows Start menu.

Configuration

Once the software is installed, the next step is to configure it with the specific information about the system e.g. what kind of slides are attached, what are the stroke-lengths of each, the I/O information etc.

Basic Configuration

Following demo shows the basic configuration of the software.

I/O Configuration

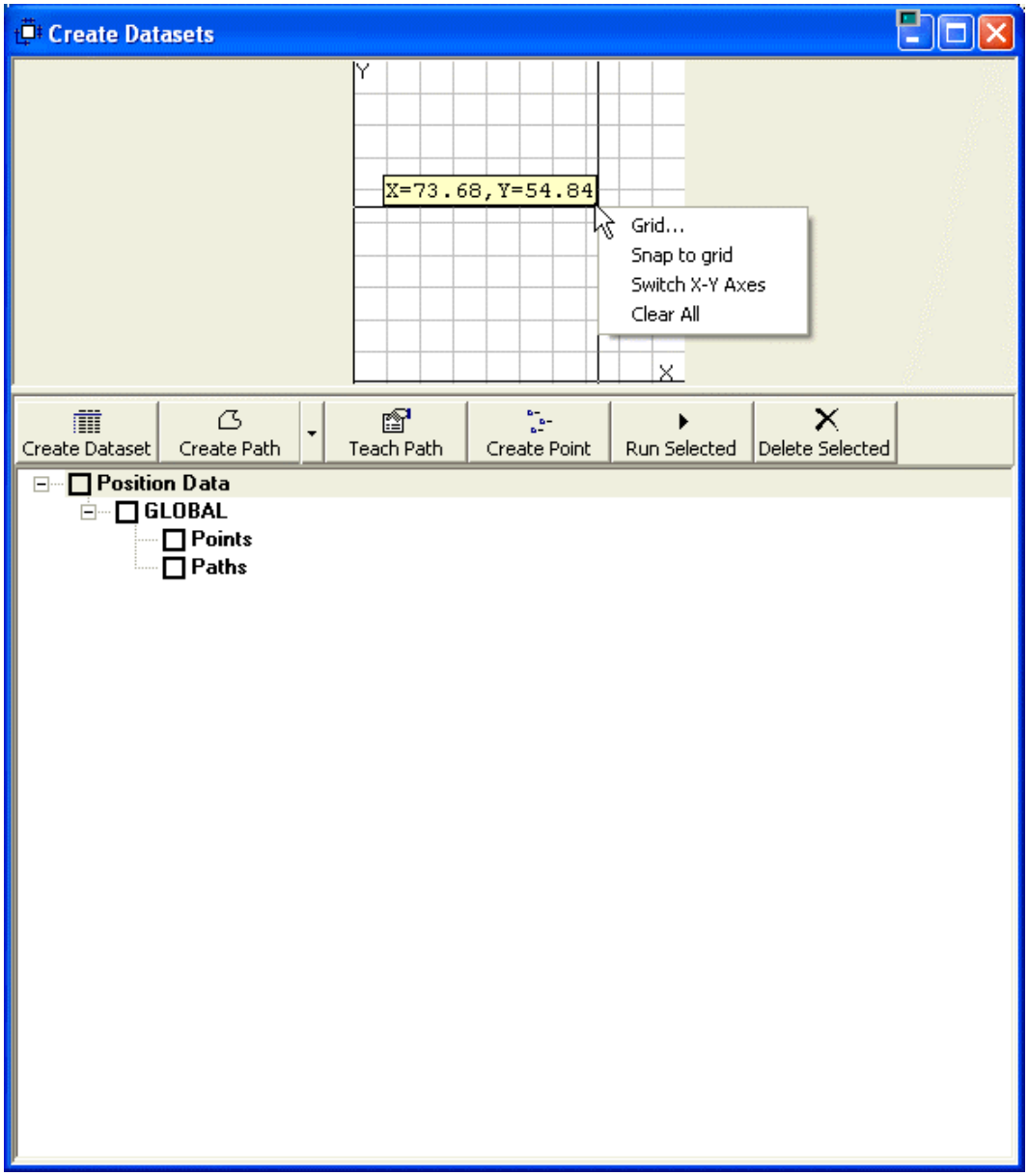
Following demo shows how to configure I/O onboard the motors as well as on the SLA External I/O Module.

Manual Operations

The SLA OS comes with powerful manual operations capability that can be used to test the slides, test the created position and path data as well as interact with the slides directly through motor terminal. Following demo shows how to use Teach Pendant, how to do homing of the motors and how to run diagnostics to automatically fix problems with slides where possible. **Note that the slides must be homed before the valid position data can be taught.** This demo shows Teach pendant operations, homing motors and running diagnostics.

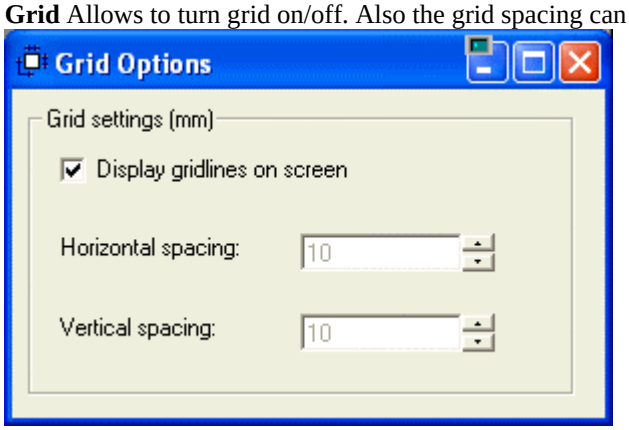
Creating Position Data

Almost all programs require a well defined position data information. Using SLA OS's visual representation of XY plane, it is very convenient to create the data in XY plane.
The main Dataset screen can be accessed by clicking on 'Datasets' icon on the toolbar.



The above picture shows the window after moving the mouse cursor on the graphical XY plane and right clicking on the region. Note the coordinates of a point are displayed while the cursor is on the region. The options available to configure the graphical area in the menu includes

1. **Grid** Allows to turn grid on/off. Also the grid spacing can be controlled from here.



2. **Snap to Grid** Once selected, the mouse pointer will snap to the grid intersections. This is very useful when positions to be created are known to be in the multiples of grid spacing.
3. **Switch X-Y Axes** Depending on the configuration of X-Y slides, the alternate orientation (Y axis horizontal and X axis vertical) can be useful in visualizing the points getting created.
4. **Clear All** This will clean the graphical screen of any drawings.

Creating Position Data

This demo shows how to create position data using graphical area for both linear and curvilinear paths and independent positions.

There are two types of position data that can be created using the window.

1. **Points** These are independent collection of points, each of them defining a location in space. They can be created using 'Create Point' button on the toolbar.
2. **Paths** These are ordered collection of points defining a trajectory in space. There are two types of path, both of which can be defined using 'Create Path' dropdown combobox. By default, clicking on 'Create Path' will create Linear path.
    1. Linear - This is made up of connected straight lines.
    2. Arc - This defines a section of circle.

This position data can be contained in a dataset created using 'Create Dataset' button. It is highly recommended that each distinct project (where data is not shared) has its own dataset. This makes it easier to organize as well as access the data during programming. Once a dataset is created, the points or the paths contained within can be accessed following <datasetName>.<pointName> for a point and <datasetName>.<pathName> for a path e.g. MyDataset.path1 or MyDataset.point1. The system already comes with 'GLOBAL' dataset for sharing position data that could have cross project usage (e.g. origin). The data contained with 'GLOBAL' dataset can be accessed with or without prefixing the name 'GLOBAL' before it. e.g. GLOBAL.<pointName> or <pointName> both will work for the points contained in the 'GLOBAL' dataset.

Creating Points

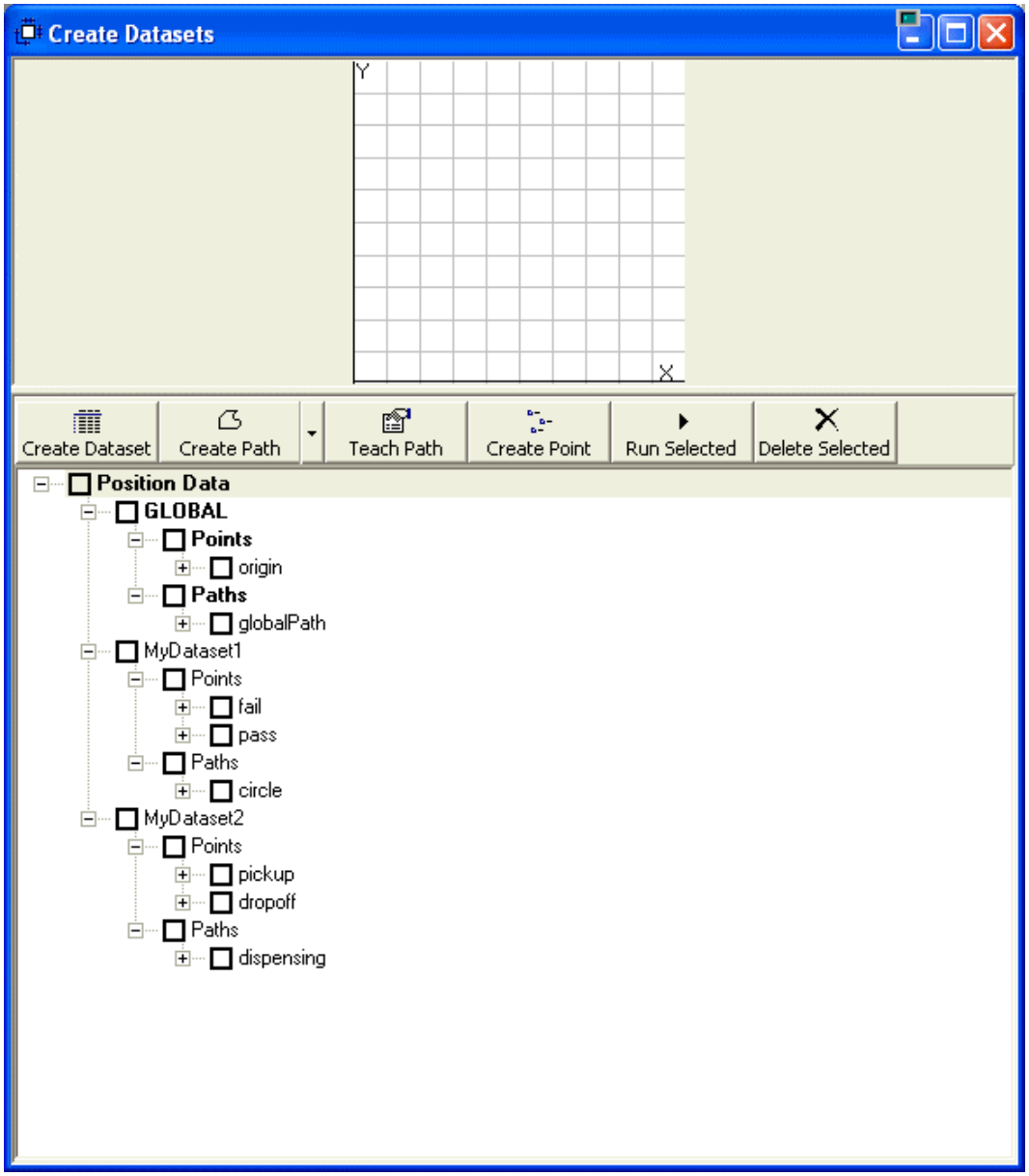
There are more than one way to create/update an independent point for a selected dataset. To create a point, select the 'Points' under the desired dataset. With the mouse, click on the desired point in the graphical area. Since the Z axis value is obtained from the current Z location, it might be beneficial to move to the desired Z location before creating points. Another way to create a point is to manually move the slides to the desired location and the clicking on 'Create Point' button on the toolbar. To make it easier to move slides by hand, be sure to turn off servo and disengage the automatic brake by clicking on 'Motor Servo' and 'Automatic Brake ON' buttons for the corresponding motors in the teach pendant tab on the main screen. This is very useful in creating points out of distinct locations in the work area. If any of the values require further

refinement, they can be edited by selecting and pressing the 'ESC' button or by clicking twice on the desired coordinate value.
This demo shows how to create position data using current motor positions and updating an existing position with the current motor position.

Creating Paths

Paths can also be created in multiple way. Before any points can be added to a path, create the correct type of path (Linear or Arc) using the 'Create Path' button. For Linear paths, select the path and create sequence of points following the same sequence used for creating individual points. Notice that as more points are added, the path is drawn on the screen. For Arc paths, select each of the special points that get created and update them either through direct editing or clicking on 'Update Point' button or clicking with mouse in the graphical area.

Typical screen might look like below after creating position data for multiple projects. The checkboxes next to each of the node in the 'Position Data' tree can be used to draw the child nodes in the graphical area. e.g. By selecting the 'Position Data' checkbox, all the data contained in all datasets will be drawn on the screen. This is very useful in visualizing the various points location and path trajectories.



Advanced ways to create paths

Import Dataset

This demo shows how to create a path by importing AutoCAD generated DXF file to create a path. Other supported formats are XLS (Microsoft Excel file e.g. data.xls) and CSV (comma separated values, a plain text file e.g. data.txt). Note that in the XLS and CSV formats, the first row is discarded and must not contain the needed data.

Teach Dataset

This demo shows how to create a path by turning the servo and brake off and manually moving the motors around to teach a path.

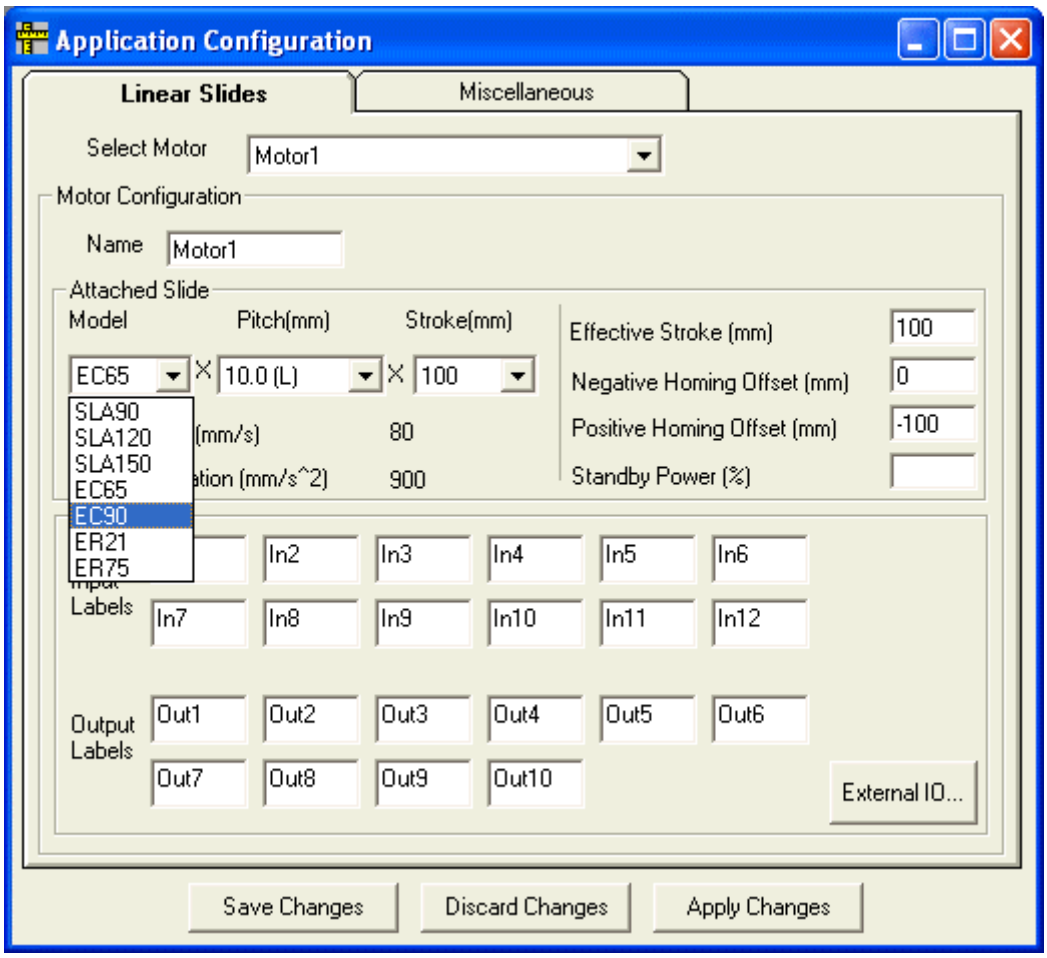
Introduction to eCylinder/eRotary

eCylinder models (EC65 and EC90) are new additions to the growing electric product lines from DE-STA-CO Automation. It augments the existing SLA90, SLA120 and SLA150 products to provide flexible deployment options. Following are the major differences and notable similarities between the two types.

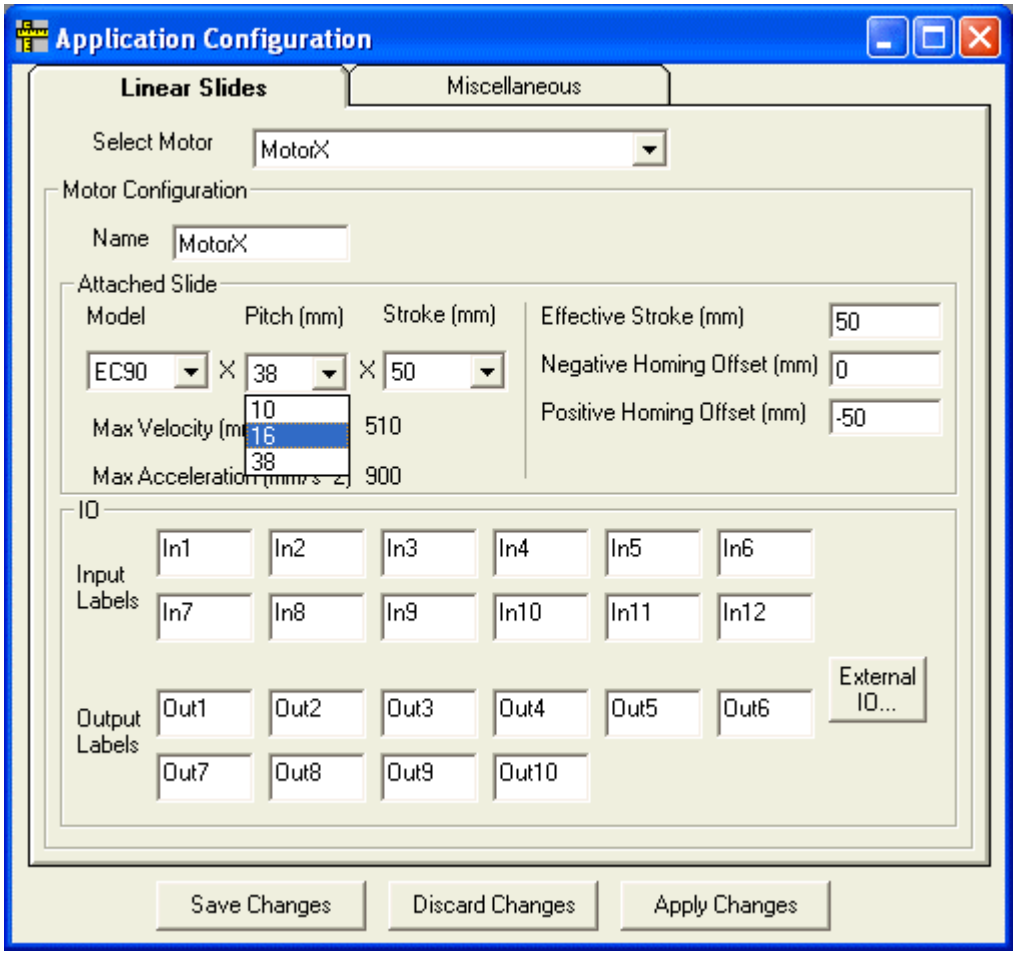
- eCylinder/eRotary are backward compatible with the Smart Motor Programming (SMP) command set which is used to drive the motors from the host computer.
- Currently only the BCD programs can be downloaded to the eCylinder/eRotary. Programming them for the BCD movements is same as that for SLA.
- eCylinder/eRotary can be operated using MMP programs. However, since they can't participate in a coordinated motion, the sla.DoLine and sla.DoPath commands are ignored by it. These motion commands can still be used to move SLA motors in a mixed deployment stage (e.g. X and Y are SLA slides, while Z is an eCylinder/eRotary). To move an eCylinder/eRotary from a MMP program, either of sla.DoPositionMove, sla.DoRelativeMove or sla.DoVelocityMove commands can be used.
- There are ten onboard outputs in eCylinder compare to six for SLA.
- eCylinder/eRotary have absolute encoders. However, to take into account the homing offsets, it is still necessary to carry out homing which adjusts the origin according to the offset values.

Configuration Changes

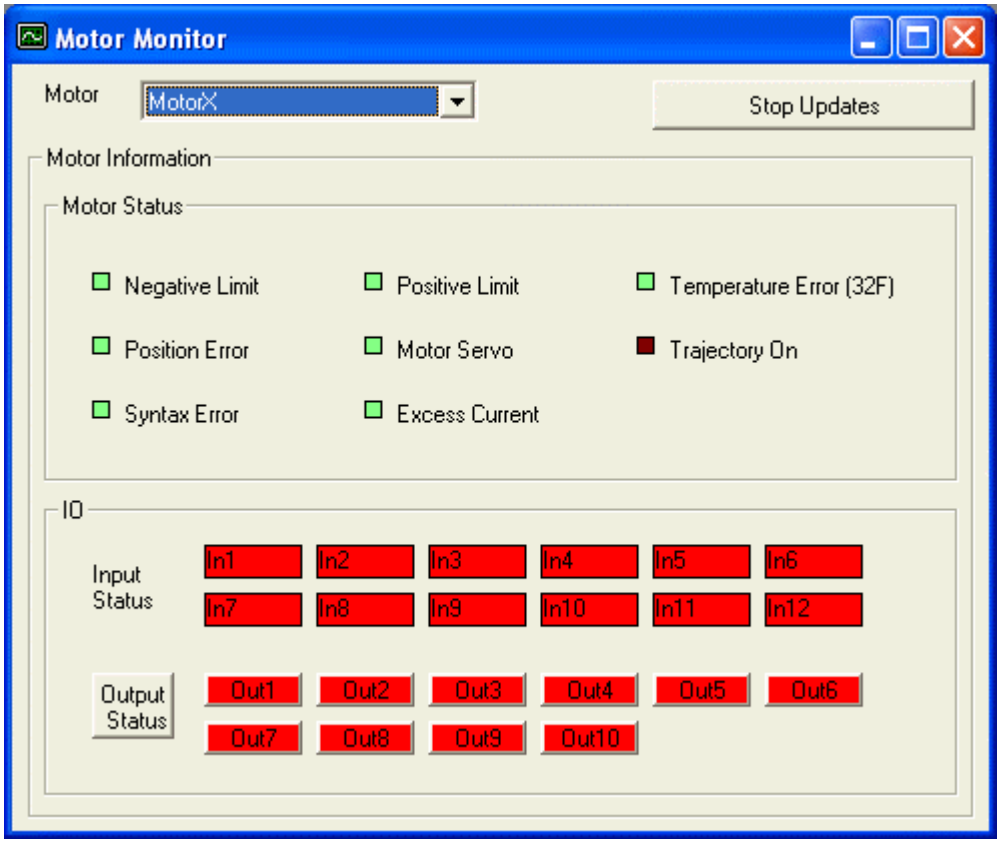
The main changes to the SLA OS program are in the configuration screen as shown below.



Two new eCylinder models (EC65 and EC90) and two new eRotary modesl (ER21 and ER75) are available for selection.



Each of the eCylinder model comes with multiple pitch selection. Also note the increased number of outputs to 10.



Motor monitor window shows correct number of IO states depending on the selected slide.

eRotary Motion Specification

While SLA/eCylinder has been well understood in terms of what direction and by how much the movement would occur, eRotary (which has no negative/positive limits) needs clarity due to its circular nature. Following results are meant to assist in understanding the eRotary motion. All angles are specified in degrees.

Following is the result of various absolute positions starting from the current locatin at origin (RP = 0)

- P=360, motor will move 360 in clockwise direction. [RP returns 0]
- P=-360, motor will move 360 in anticlockwise direction. [RP returns 0]
- P=90, motor will move 90 in clockwise direction. [RP returns 90]
- P=-270, motor will move 270 in anticlockwise direction. [RP returns 90]
- P=450, motor will move 450 in clockwise direction. [RP returns 90]
- P=-450, motor will move 450 in anticlockwise direction. [RP returns 270]

Following is the result of various absolute positions starting from the current locatin at 90 (RP = 90)

- P=360, motor will move 270 in clockwise direction. [RP returns 0]
- P=-360, motor will move 450 in anticlockwise direction. [RP returns 0]
- P=90, motor will move 0 in clockwise direction. [RP returns 90]
- P=-270, motor will move 360 in anticlockwise direction. [RP returns 90]
- P=450, motor will move 360 in clockwise direction. [RP returns 90]

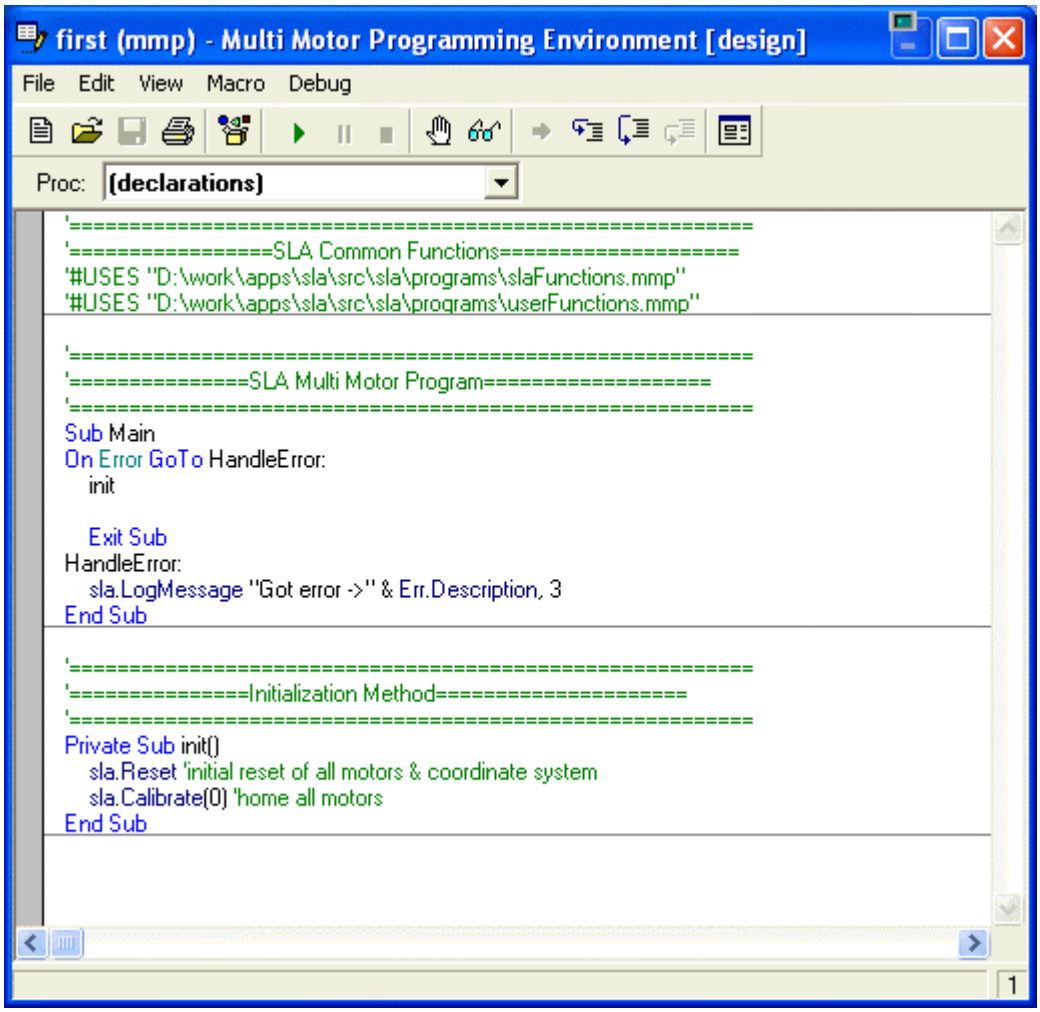
- P=-450, motor will move 540 in anticlockwise direction. [RP returns 270]

Programming

Creating Multi Motor Programs (MMP)

Multi Motor Programs (MMP) are the easiest and most flexible of all the programs available for SLA OS. Since these programs run on the PC Controller, they can create coordinated motions (by coordinating each of the slides), communicate with external IOs, interface with other USB/Ethernet devices and offer a very powerful development environment for creating and debugging programs in easy to use Basic language.
Following demo shows main steps for a simple MMP program.

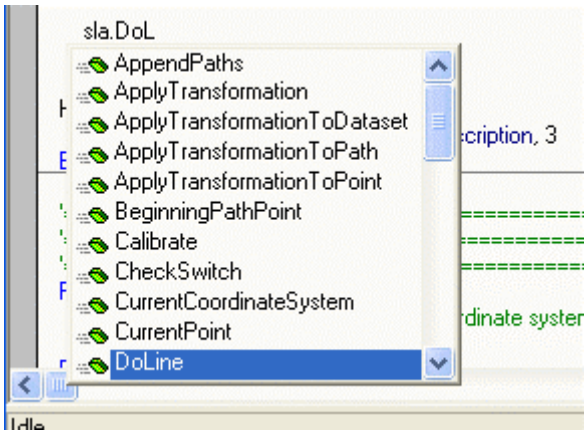
To create a MMP program, open a 'New Program...' dialog by clicking on File menu or pressing Ctrl+N (Control and N together). By default, the MMP program is selected. Type the name of the new program in 'Program Name' textbox and click 'Create Program' button. This will bring up the following window for developing MMP program with the template of the program already created as shown below.



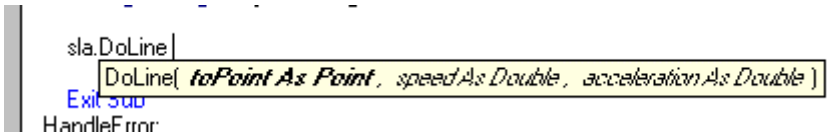
Type following text just after the init statement.

```
sla.LogMessage "My First Program"
```

Save the program by selecting 'Save' from the File menu or clicking the save icon in the toolbar. For running the program, click on the run icon or press 'F5'. (Caution: since by default the program will home the motors, make sure that there is no body near the slides or no obstacles are placed which can prevent homing. If desired, homing can be disabled by commenting out the 'sla.Calibrate(0)' statement in the init subroutine.). This should create a log entry in the 'Logs' tab of the main screen as well as writing in the logs.txt file located in the logs directory where the software is installed. All the commands that interface with the SLA OS start with 'sla.' prefix. As soon as 'sla.' is typed in the editor, the intellisense facility shows all the commands that available. The intellisense facility can also be activated by hitting 'CTRL-SPACE'.



Select the command that is appropriate for the use. The arguments to the command are also displayed as soon as a SPACE or '(' is typed.

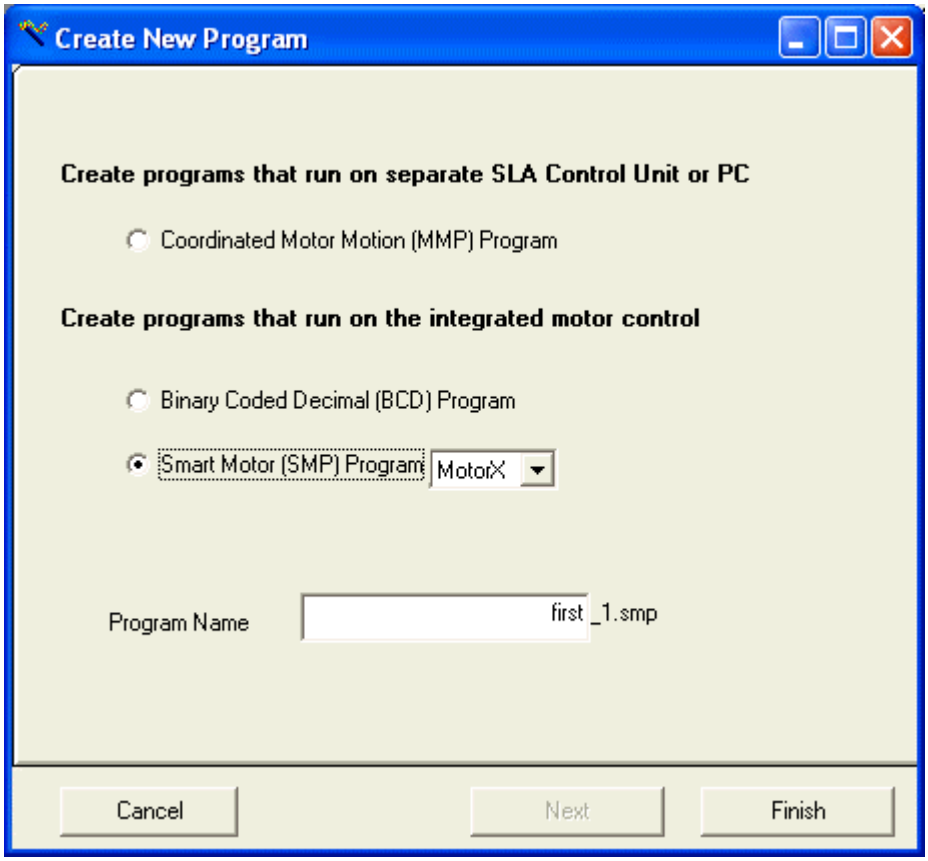


This makes it very convenient to write a MMP program.

Creating Smart Motor Programs (SMP)

Smart Motor Programs (SMP) are normally created when there is no need for coordinated motion and the (optional) SLA Control Unit is not present. These programs can be created by using any supported Windows platform and uploaded directly into the motors. When the power is turned on, these programs run independently of each other in the individual motors. However, using shared IO signals some parts of these otherwise independent motion can be synchronized by waiting on the common signal. Since there is no real time coordination among the motors for a well defined trajectory, the resulting movement is sometimes referred as synchronized motion. The SMP programs are developed using its own development environment which allows for syntax coloring, syntax checking, compiling and easy uploading and downloading of the programs to a motor. These programs are written using simple Smart Motor Language.
Following demo shows main steps for a simple SMP program.

To create a SMP program, open a 'New Program...' dialog by clicking on File menu or pressing Ctrl+N (Control and N together). Select the SMP program type and enter the name of the new program in 'Program Name' textbox. Note that by default, MotorX is selected. If you wish to write a SMP program for different motor, change the selection. Click 'Create Program' button.



This will bring up the following window for developing SMP program with the template of the program already created as shown below.

Study the generated template carefully, as it exhibits many of the common programming techniques used in SMP programming including using variables, issuing SMP commands, using flow control techniques, writing subroutines, invoking subroutines, conditional expressions etc. By default, the generated template contains the code for downloading the tuning values and invoking homing subroutine at the power startup. This behavior can be changed by directly

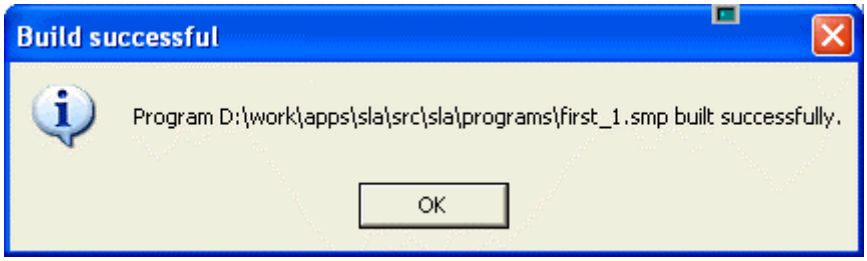
modifying the program in the editor. Enter the following program in the area indicated for user code.

```
MX
a = 1
WHILE a < 10
  a = a + 1
  D = 1000
  V = 100000
  A = 100
  G
  TWAIT
LOOP
```

While at a first glance, it looks like a valid program for performing relative movement for 10 iterations, let's try to build the program to see if all the syntax is correct. Click on the 'Build the Smart Motor Program' icon in the toolbar. Since the first command MX doesn't exist, you will get an error (automatically highlighted in red) and the cursor is moved to the error line as shown below.

To correct the error, simply change the command from MX to MP for position mode of motion.

Try building the program again. This time it should build correctly and a dialog informing successful build is displayed.



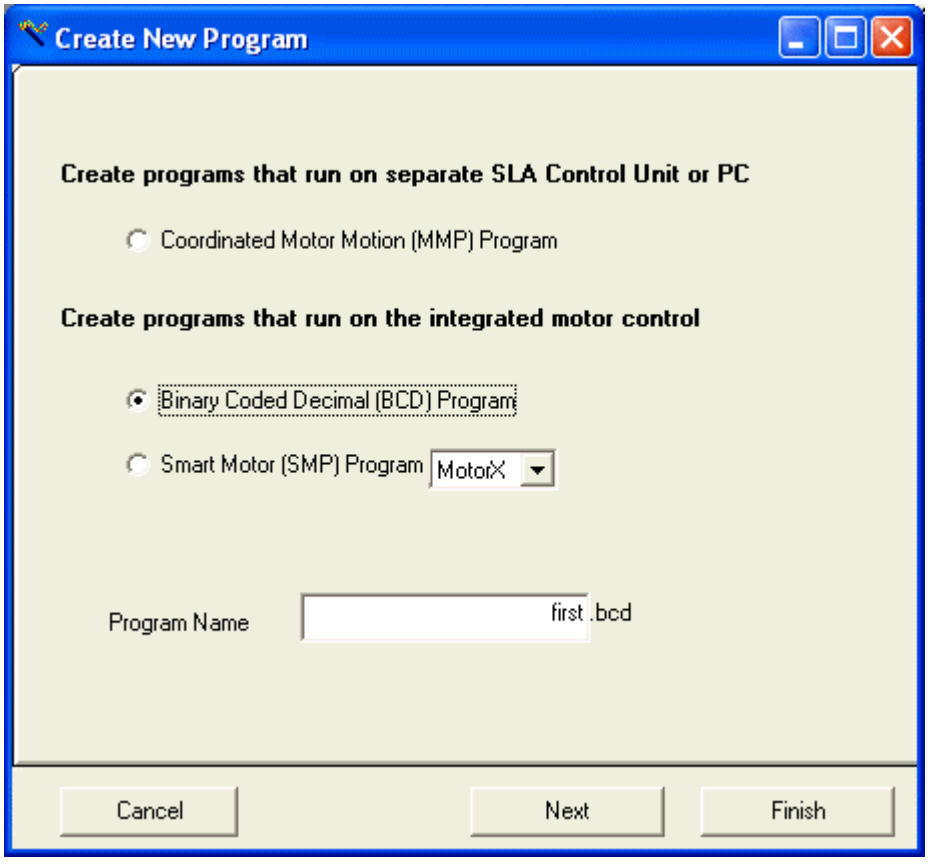
To run the program, upload in the motor by clicking on 'Transfer Program from PC to Motor' on the toolbar and clicking the run button. This will result in ten short movements in the positive direction for MotorX.

The SLA OS gives you complete control over the write, build and test process and you can iterate the process as many times as required to make sure that the program meets all your requirements. Once satisfied with the program, the PC is no longer required. When the power is turned ON again for the motor, the program will automatically start running from the beginning (by default homing).

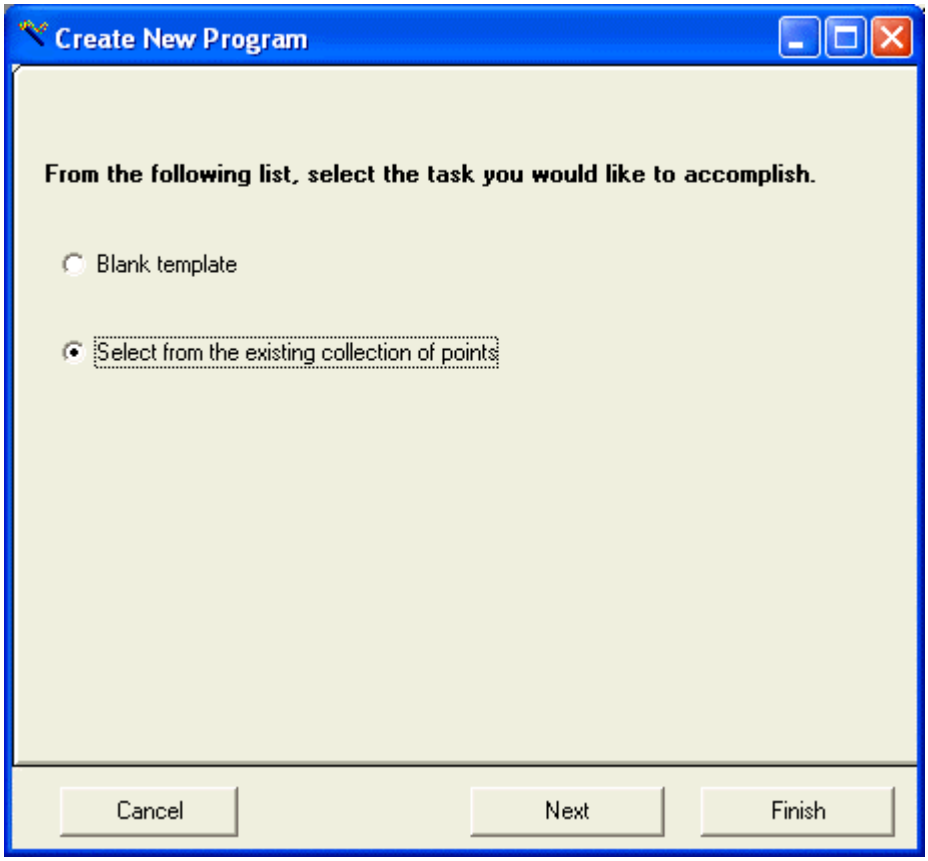
Creating Binary Coded Decimal Programs (BCD)

Binary Coded Decimal Programs (BCD) are nothing but simplified SMP programs. Since PLC based controls are very effective in coordinating various devices and generally used to move the slides to preprogrammed locations depending on the combination of BCD bits, this category of SMP programs have been separated out for special treatment. For BCD programs, all of the programming code is auto generated based on the selection of the locations. This includes tuning parameters download and homing subroutine. Since the generated code is a valid SMP program, it can be opened and edited in the much more flexible SMP programming environment. This is sometimes necessary when more custom actions are required to achieve the goal.
Following demo shows main steps for a simple BCD program.

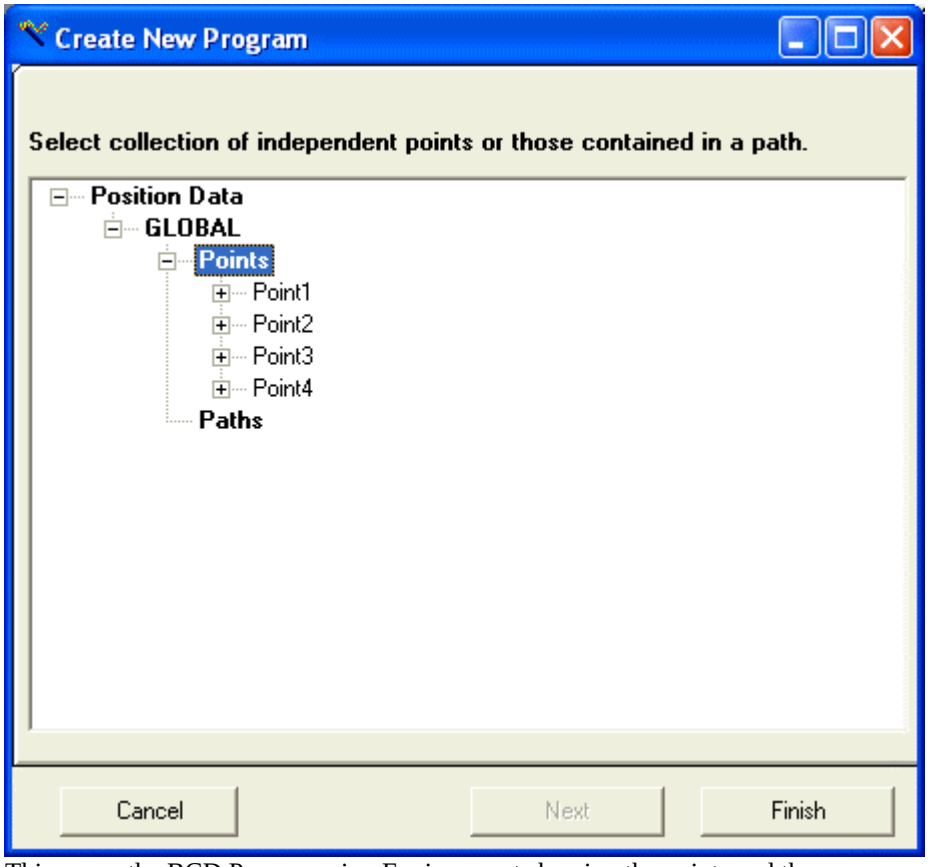
Before creating a BCD program, create few points as shown in the BCD Quick Start earlier. Now, to create a BCD program, open a 'New Program...' dialog by clicking on File menu or pressing Ctrl+N (Control and N together). Select the BCD program type and enter the name of the new program in 'Program Name' textbox. Click 'Next' button.



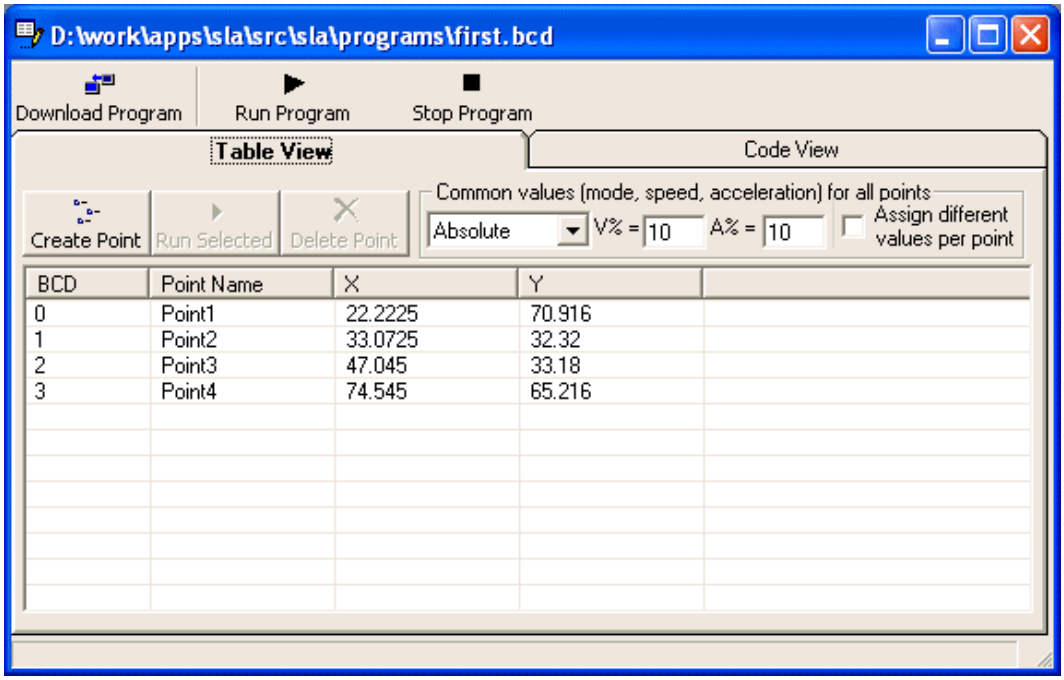
In the task selection window, you can select different ways to create a BCD program. Select the second option for creating a BCD program from existing data.



This will bring up a window with the position data tree that was earlier created in the Datasets window. Select the collection of points (either a path or points) to create BCD program from as shown below.



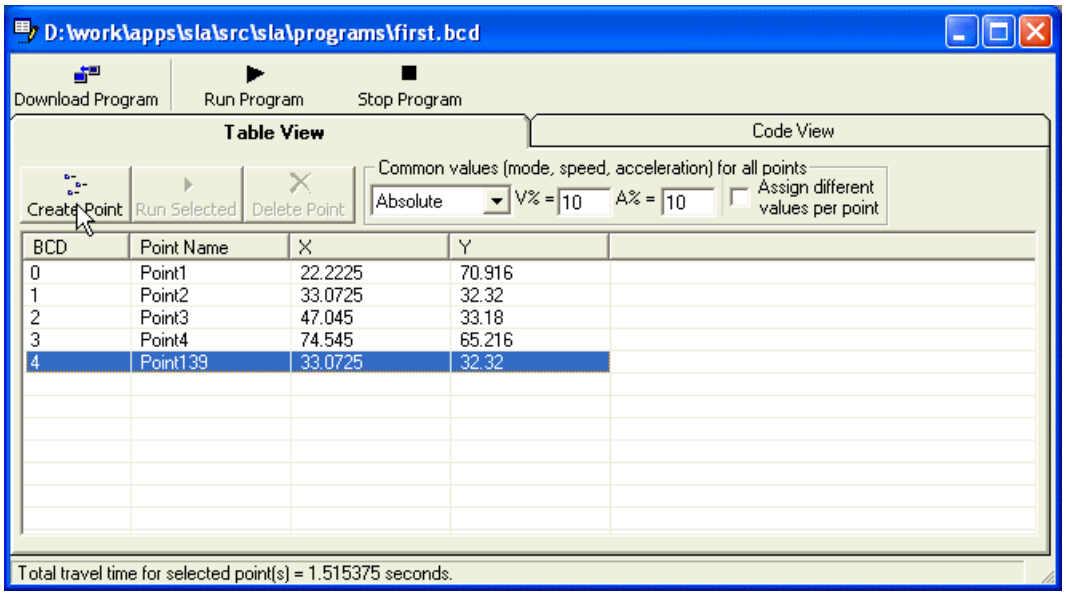
This opens the BCD Programming Environment showing the points and the corresponding BCD values.



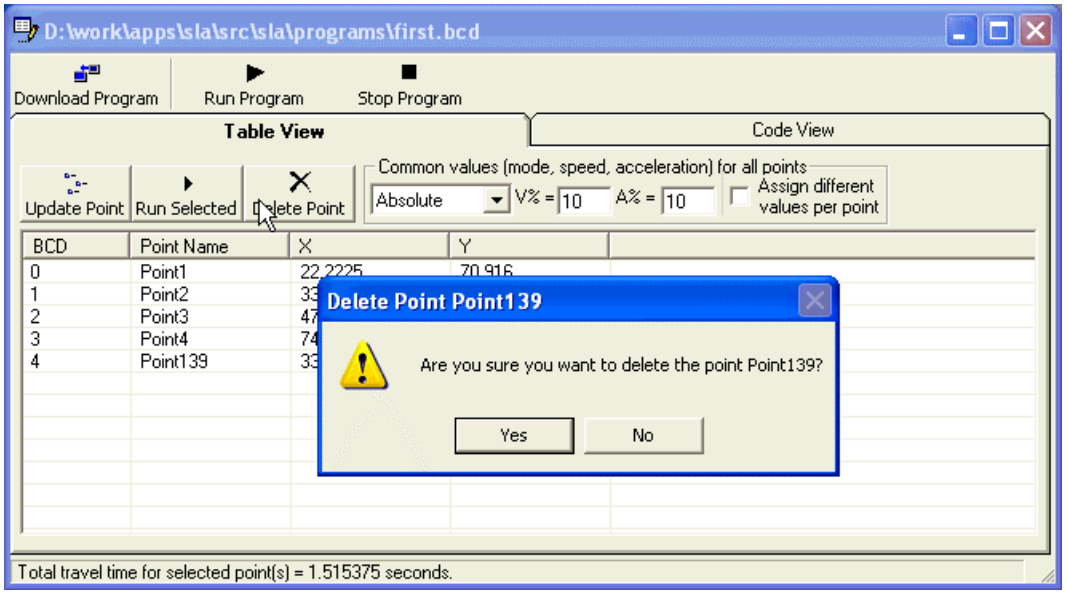
Note the automatically assigned BCD values (starting from 0) depending on the order of the points in the selected collection of points. While this is the default behavior, it can be modified by using the SMP programming environment. The BCD programming environment has evolved into a

powerful IDE to create PLC driven onboard programs without writing single line of code. From this screen you can take following actions which modifies the actual generated code.

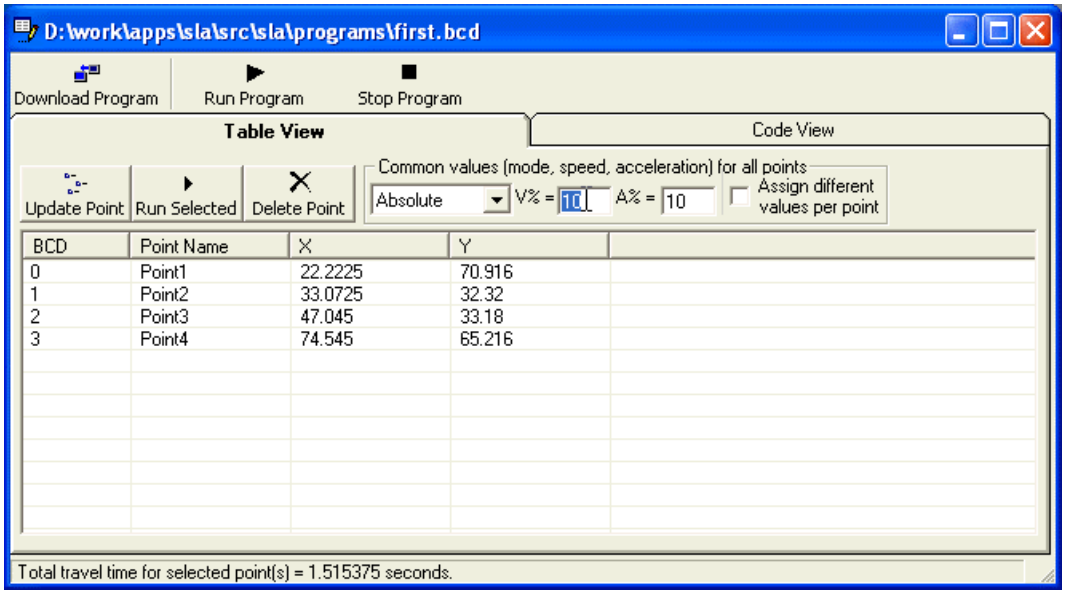
- Create an additional point in the BCD sequence.



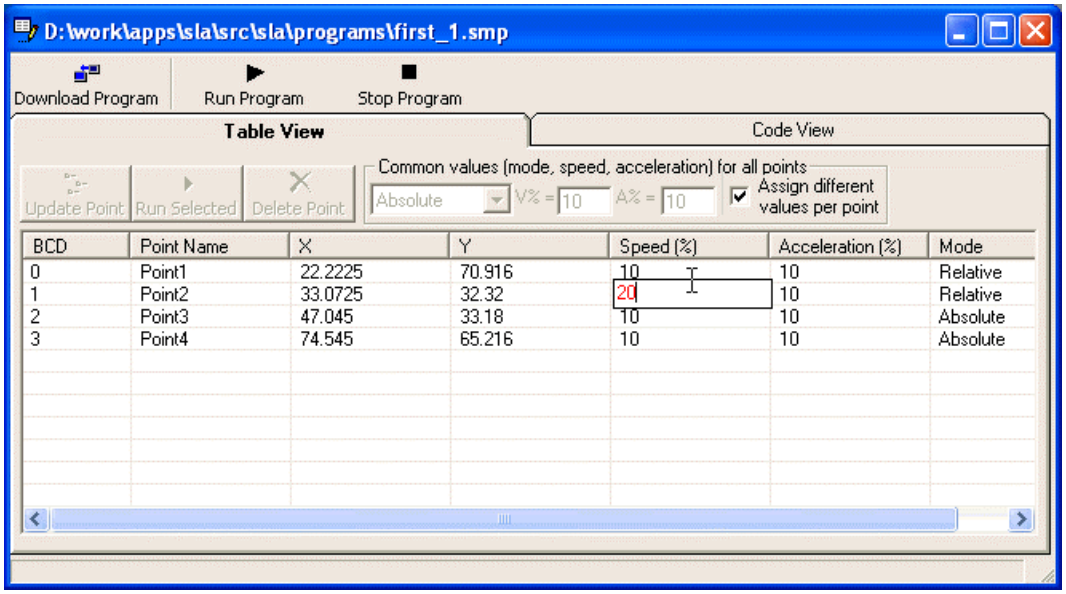
- Delete an existing point the BCD sequence.



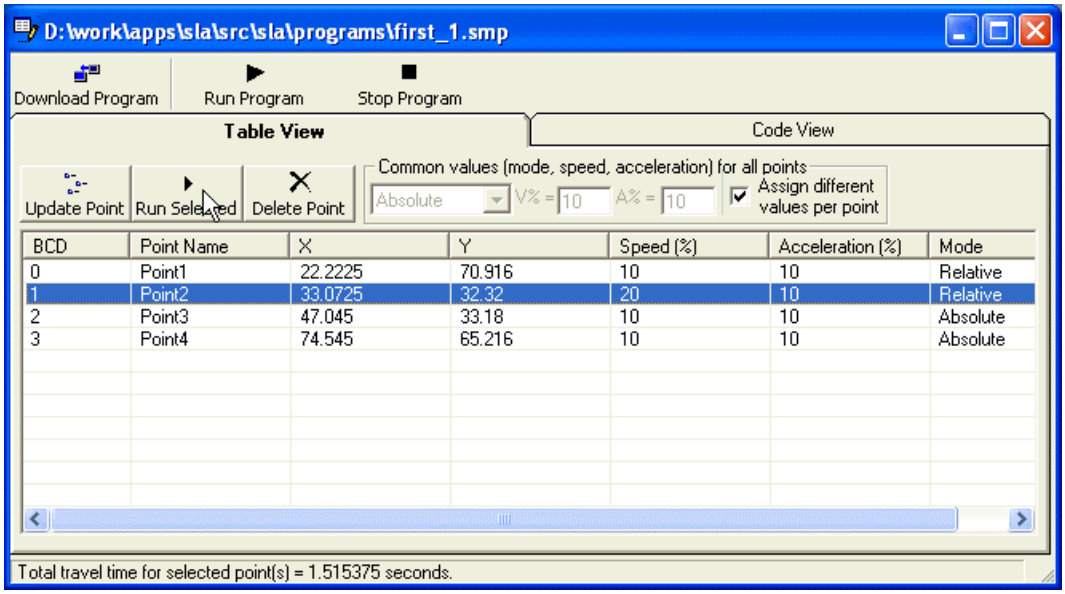
- Adjust speed, acceleration and mode (whether absolute or relative) for all points during motion



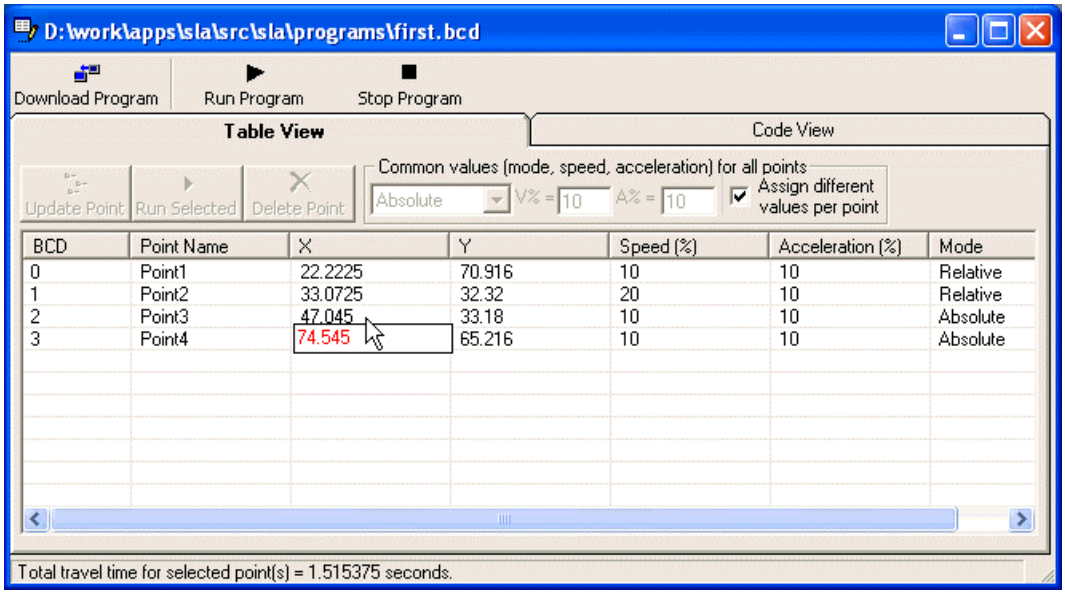
- Adjust speed, acceleration and mode (whether absolute or relative) for each individual point separately during motion



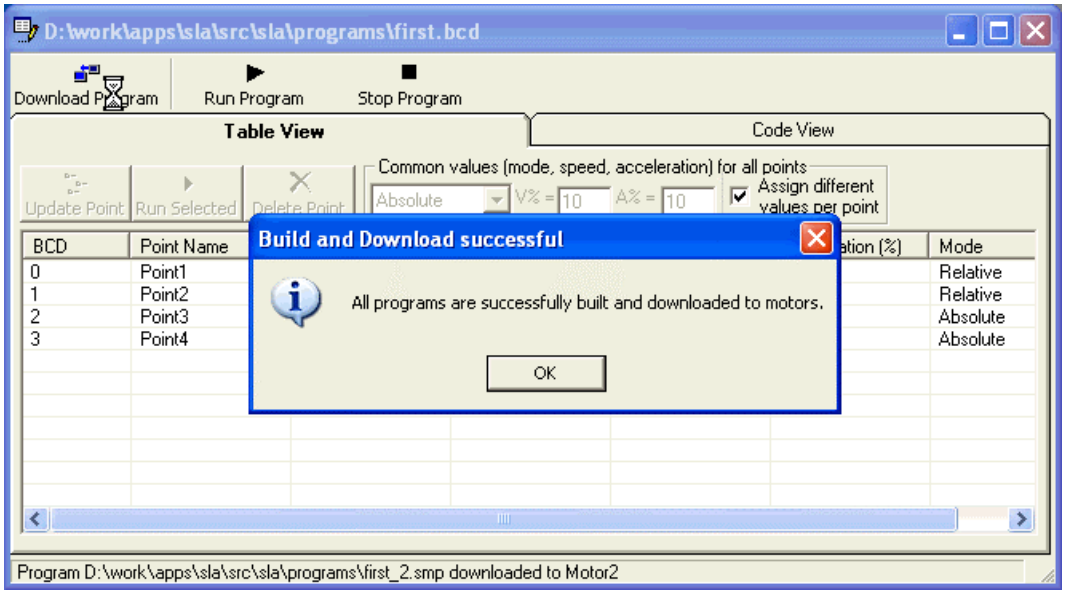
- Test any of the point by selecting and clicking on 'Run Selected' button



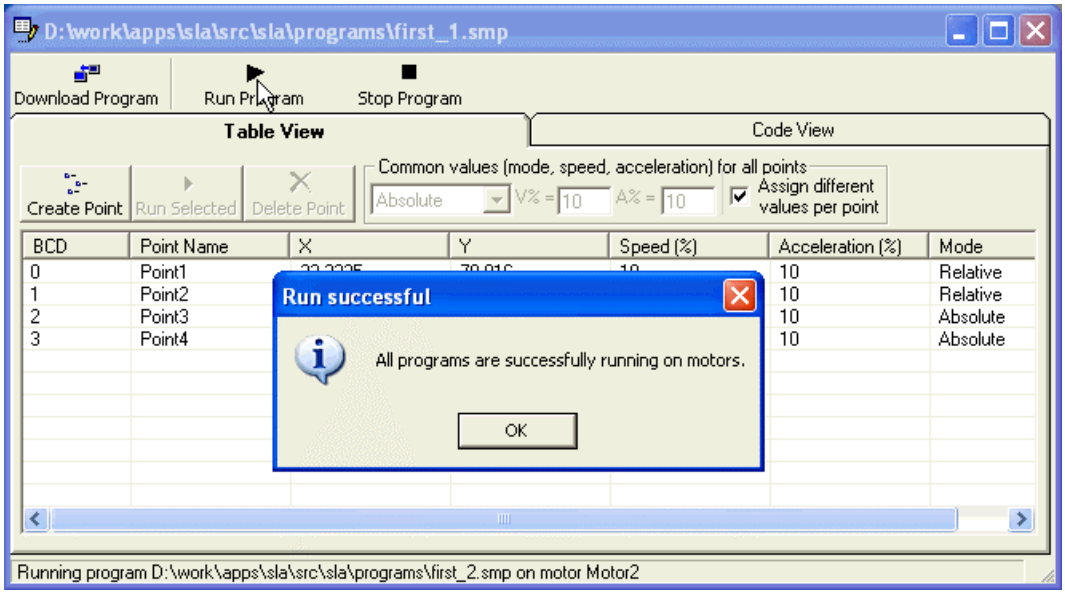
- Edit name, position values, speed, acceleration or mode directly in the table



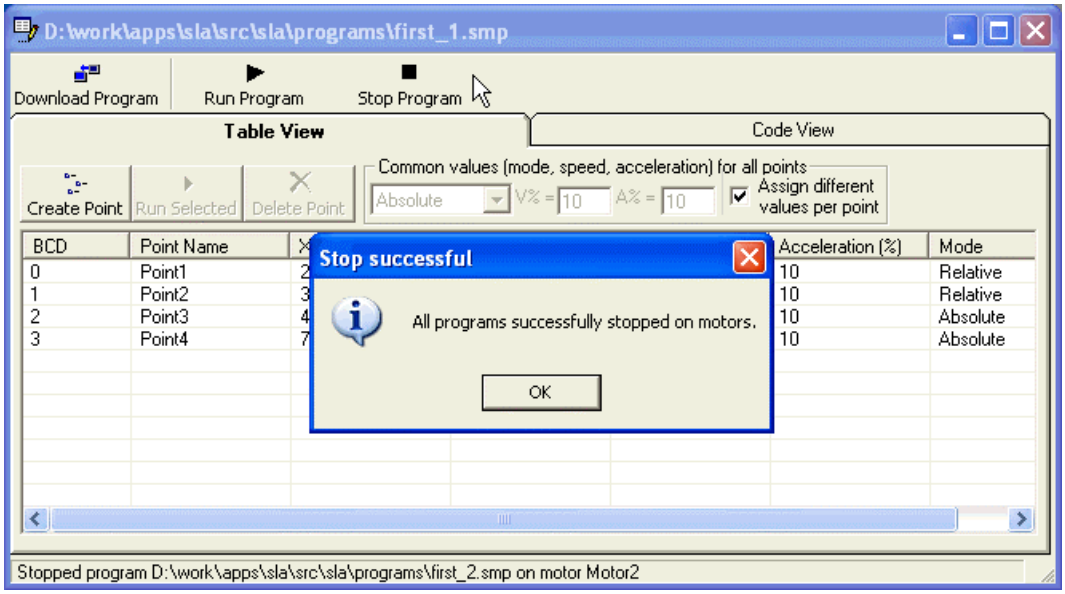
- Download generated programs to all motors with single click



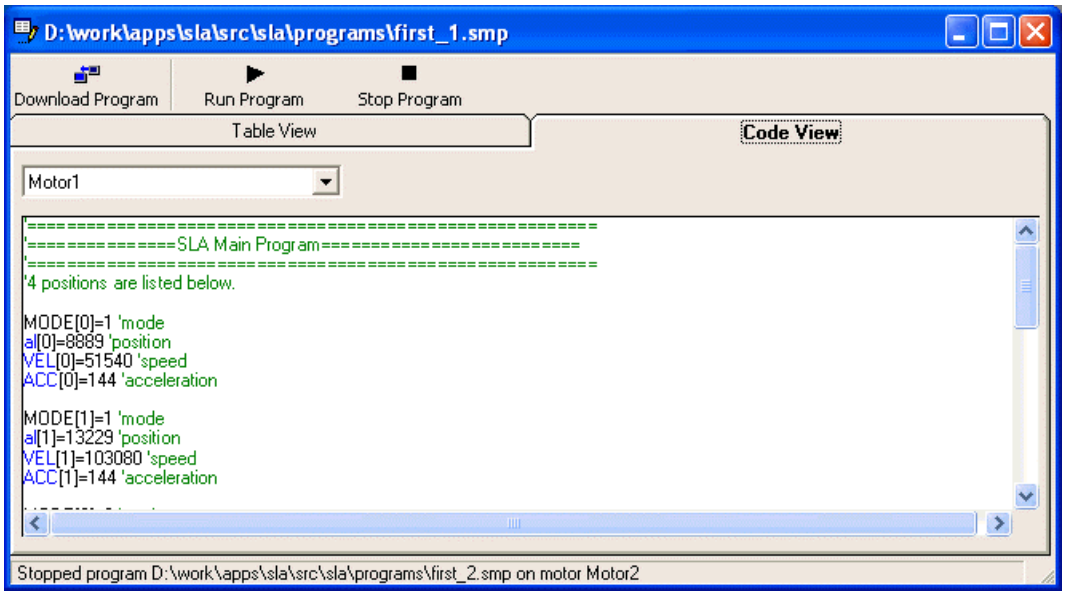
- Run all programs on all motors with single click



- Stop all programs on all motors with single click



While all the above actions are possible in the table view, code is transparently kept in synch with the table view. To view the code, you can switch to the 'Code View' as shown below. Note that the name of the corresponding generated SMP file is displayed on the title bar. Since one BCD program corresponds to as many SMP programs as the number of motors present, you can view each of those programs by selecting the appropriate motor from the dropdown box. Another thing to note is that the code displayed is not editable and the only way to modify the default generated behavior is using an SMP editor. Note that the code depends on the type of slide connected to the axis. The following display corresponds to an eCylinder.



To run the program, simply build and download the generated programs to all the motors by clicking on the 'Download Program' button in the toolbar. Once all the programs have been downloaded, click the run button to run all programs.

External Systems

SLA OS can connect to the external vision systems to access the image analysis data. This data can be used in a MMP program to drive the SLA system.

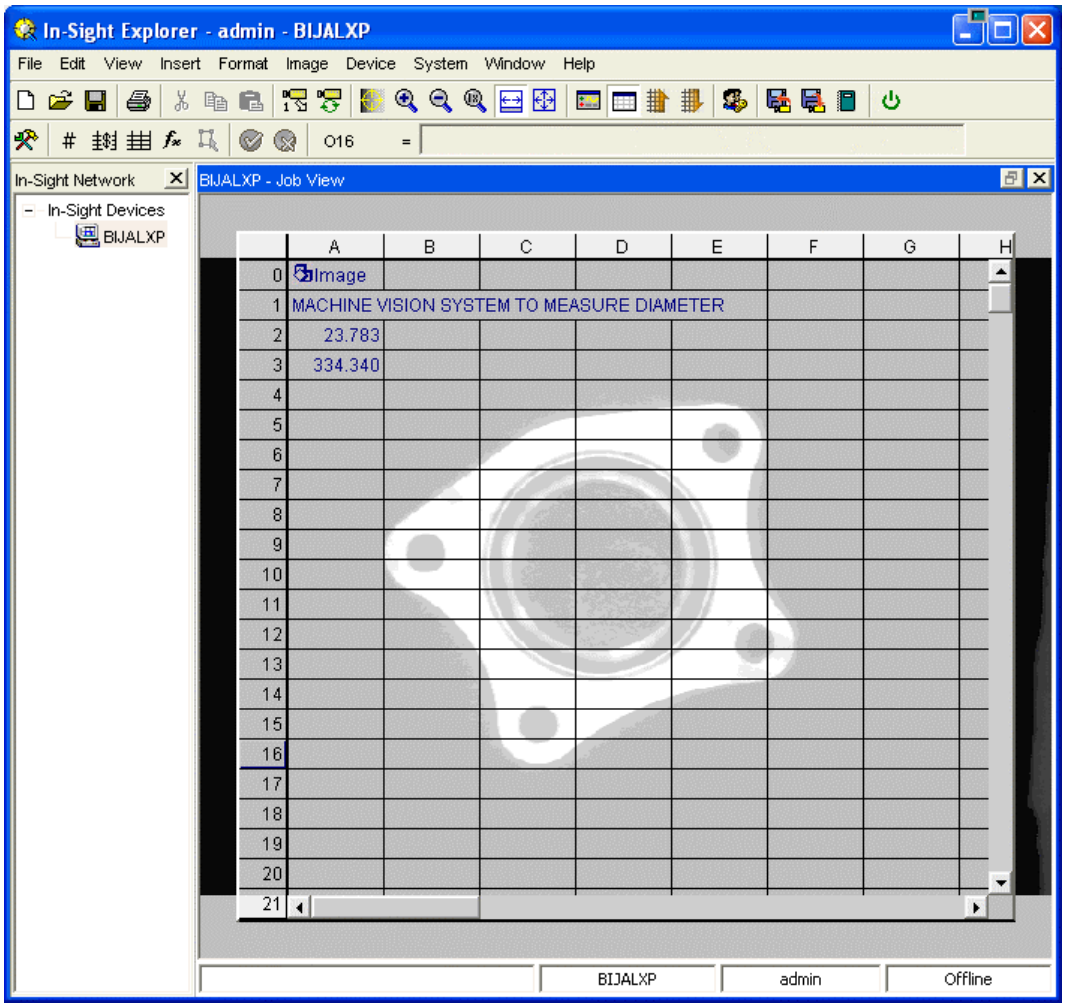
Following Vision systems are supported in the SLA software.

- Cognex Machine Vision
- DVT Machine Vision (coming soon)

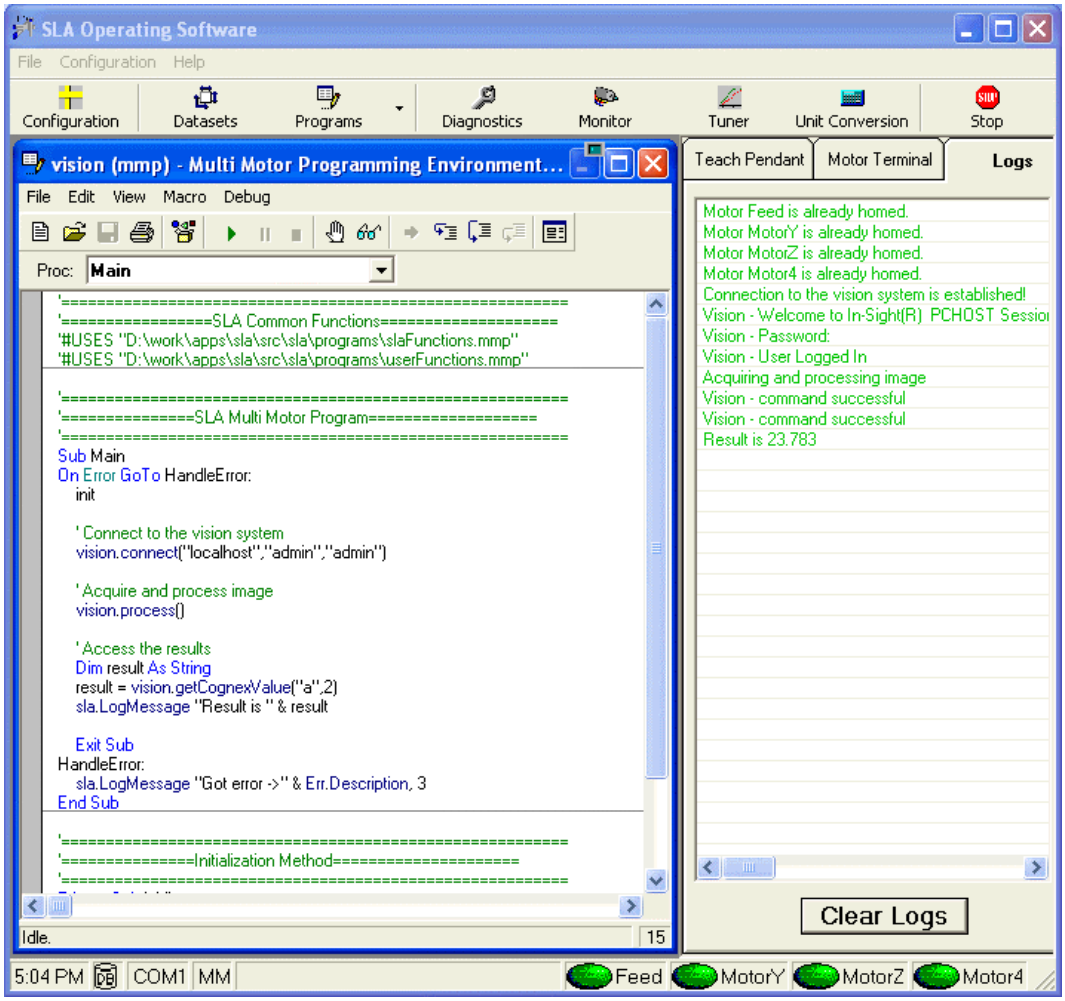
Using SLA OS's vision interface, it is trivial to connect to the supported vision systems. The three main steps involved are:

1. **Connect** - First step is to connect to the vision system. The underlying software takes care of all the native connection details.
2. **Process** - Once connected to the vision system, the image acquisition and processing by the vision system can be triggered programmatically, for example, depending on the selected input signal.
3. **Access** - The results of the processing of the machine vision image by the camera can be accessed using simple access command.

Following image shows a Cognex device after acquiring the image and processing to show the result value of 23.783 in A2 cell.



Following simple program shows the three steps mentioned earlier to access the result value. The log shows the successful reading of the value.



Once the value is read, subsequent steps in the program can depend on the analysis of the results.

For complete descriptions of the vision commands, please refer to the Vision Commands Reference.

Examples

Sample programs

Following is a list of programs that are packaged with the SLA OS software and can be found in the *samples* folder under the main installation location (usually C:\Program Files\Robohand\SLA). Most of these programs also have the corresponding database files which contains the data about points and paths used in the programs. To run a program

1. Copy the program file in the *programs* folder under the main installation location.
2. Copy the corresponding database file in the *data* folder under the main installation location.
3. Start the SLA OS program and select the correct database from the dropdown list of databases.
4. Select the corresponding program from the File/Open menu.
5. If it is SMP or BCD program, build, download and run the program. MMP program can be run from the host controller itself.

MMP programs

- **MMP-Prog1-HOMING** [source]
  This is a basic example of homing triggered by an user input.
- **MMP-Prog2-MessageWindow** [source]
  This program demonstrates blocking message window with acknowledgement.
- **MMP-Prog3-MovingPointToPoint** [source]
  This program demonstrates simple point to point coordinated motion.
- **MMP-Prog4-LinearPath** [source]
  This program demonstrates basic linear path coordinated motion with various parameters.
- **MMP-Prog5-CurvilinearPath** [source]
  This program demonstrates basic curvilinear path coordinated motion with various parameters.
- **MMP-Prog6-AppendedPath** [source]
  This program combines various paths into a single path for creating complex composite path.
- **MMP-Prog7-ControllingIOs** [source]
  This program demonstrates various commands for controlling IOs.
- **MMP-Prog8-CreatingNewPoint** [source]
  This program creates a new point dynamically based on user inputs.
- **MMP-AdvancedProg1-ControllingTorque** [source]
  This is an advance example for reducing maximum current output to the motor which in turn limits the maximum torque output by the motor.
- **MMP-AdvancedProg2-Autostart** [source1, source2]
  This is an example of multi tasking feature with one program monitoring the health of the system and launching the main application program when system is ready.
- **MMP-AdvancedProg3-OffsetPattern** [source]
  This is an example of utilizing advanced transformation commands to offset a path along various axes.
- **MMP-AdvancedProg4-RotatePattern** [source]
  This is an example of utilizing advanced transformation commands to rotate a path along an axis at various angles.

SMP programs

- **SMP-Prog1-HOMING** [source1, source2]
  This SMP program shows how to control IOs and wait for user input before proceeding with homing.
- **SMP-Prog2-JOGGING** [source]
  This program shows how to create Teach Pendant like functionality using a toggle switch and a push button.
- **SMP-Prog3-PICK-N-PLACE** [source1, source2]
  The two independent SMP programs use common IOs to synchronize their movements.

Reference

MMP Commands Reference

Following are the MMP Commands which can be used in a Multi Motor Program (MMP) along with SLA and HMI Commands. They are arranged in related groups and in alphabetic order for easy access.

MMP Commands in Related Groups

The MMP Commands are categorized in each group depending on their functionalities.

- **MMP Declaration Commands**

  These commands are used to declare the variables, functions and subroutines in a program.

  - Const Definition
  - Dim Definition
  - Function Definition
  - Main Sub
  - Option Definition
  - ReDim Instruction
  - Sub Definition

- **MMP Assignment Commands**

  These commands are used to assign values or objects to the variables.

  - Set Instruction

- **MMP Flow Control Commands**

  Using flow control commands, the path of execution through a program can be controlled by checking for values of variables or IO switches. They are also used to perform iterations using loop constructs.

  - Do Statement
  - End Instruction
  - Exit Instruction
  - For Statement
  - For Each Statement
  - Goto Instruction
  - If Statement
  - Select Case Statement
  - While Statement

- ## MMP Error Handling Commands

  For a robust program, error handling can be achieved by using the On Error command.

  - Err Object
  - On Error Instruction

- ## MMP Conversion Commands

  These commands can be used to convert one datatype to another.

  - Array Function
  - CStr Function

- ## MMP Variable Info Commands

  These commands give additional information about variables.

  - LBound Function
  - UBound Function

- ## MMP Math Commands

  These commands can be used for mathematical calculations.

  - Abs Function

- ## MMP String Commands

  These commands relate to the text string handling capability of a program.

  - Str$ Function

- ## MMP User Input Commands

  Using these commands, a dialog box can be generated to display or obtain information.

  - InputBox$ Function

- MsgBox Instruction/Function

- **MMP Miscellaneous Commands**

  These are miscellaneous commands useful for programming.

  - DoEvents Instruction
  - Wait Instruction

- **MMP Operator Commands**

  These are commands related to operation between variables.

  - Operators

MMP Commands in Alphabetic Order

1. Abs Function
2. Array Function
3. Const Definition
4. CStr Function
5. Dim Definition
6. Do Statement
7. DoEvents Instruction
8. End Instruction
9. Err Object
10. Exit Instruction
11. For Statement
12. For Each Statement
13. Function Definition
14. Goto Instruction
15. If Statement
16. InputBox$ Function
17. LBound Function
18. Main Sub
19. MsgBox Instruction/Function
20. On Error Instruction
21. Operators
22. Option Definition
23. ReDim Instruction
24. Select Case Statement
25. Set Instruction
26. Str$ Function
27. Sub Definition
28. UBound Function

29. Wait Instruction
30. While Statement

MMP Commands

1. Abs Function

Summary Return the absolute value. Parameter Description Num Return the absolute value of this numeric value. If this value is Null then Null is returned.

Declaration Abs(Num)

Example

```
Sub Main
    Debug.Print Abs(9)  ' 9
    Debug.Print Abs(0)  ' 0
    Debug.Print Abs(-9) ' 9
End Sub
```

Group MMP Math Commands

2. Array Function

Summary Return a variant value array containing the exprs.

Declaration Array([expr[, ...]])

Example

```
Sub Main
    X = Array(0,1,4,9)
    Debug.Print X(2) ' 4
End Sub
```

Group MMP Conversion Commands

3. Const Definition

Summary Define name as the value of expr. The expr may be refer other constants or built-in functions. If the type of the constants is not specified, the type of expr is used. Constants defined outside a Sub, Function or Property block are available in the entire macro/module. Private is assumed if neither Private or Public is specified.

Declaration [| Private | Public] _ Const name[type] [As Type] = expr[, ...]

Example

```
Sub Main

    Const Pi = 4*Atn(1), e = Exp(1)

    Debug.Print Pi ' 3.14159265358979

    Debug.Print e  ' 2.71828182845905

End Sub
```

Group MMP Declaration Commands

4. CStr Function

Summary Convert to a string. Parameter Description Num|$ Convert a number or string value to a string value.

Declaration CStr(Num|$)

Example

```
Sub Main

    Debug.Print CStr(Sqr(2)) '"1.4142135623731"

End Sub

Date Data Typef

Group: Data Type

Description:

A 64 bit real value. The whole part represents the date, while
the fractional part is the time of day. (December 30, 1899 = 0.)
Use #date# as a literal date value in an expression.
```

Group MMP Conversion Commands

5. Dim Definition

Summary Dimension var array(s) using the dims to establish the minimum and maximum index value for each dimension. If the dims are omitted then a scalar (single value) variable is defined. A dynamic array is declared using ( ) without any dims. It must be ReDimensioned before it can be used.

Declaration Dim [WithEvents] name[type][([dim[, ...]])][As [New] type][, ...]

Example

```
Sub DoIt(Size)
Dim C0,C1(),C2(2,3)
ReDim C1(Size) ' dynamic array
C0 = 1
C1(0) = 2
C2(0,0) = 3
Debug.Print C0;C1(0);C2(0,0) ' 1 2 3
End Sub

Sub Main
DoIt 1
End Sub
```

Group MMP Declaration Commands

6. Do Statement

Summary

Declaration Do statements Loop -or- Do {Until|While} condexpr statements Loop -or- Do statements Loop {Until|While} condexpr

Example

```
Sub Main
    I = 2
    Do
        I = I*2
```

```
    Loop Until I > 10

    Debug.Print I ' 16

End Sub
```

Group MMP Flow ControForm 1: Do statements forever. The loop can be exited by using Exit or Goto. Commands

7. DoEvents Instruction

Summary This instruction allows other applications to process events.

Declaration DoEvents

Example

```
Sub Main

    DoEvents ' let other apps work

End Sub

Double Data Type

Group: Data Type

Description:

A 64 bit real value.
```

Group MMP Miscellaneous Commands

8. End Instruction

Summary The end instruction causes the macro to terminate immediately. If the macro was run by another macro using the MacroRun instruction then that macro continues on the instruction following the MacroRun.

Declaration End

Example

```
Sub DoSub

    L$ = UCase$(InputBox$("Enter End:"))

    If L$ = "END" Then End

    Debug.Print "End was not entered."
```

```
End Sub


Sub Main
    Debug.Print "Before DoSub"
    DoSub
    Debug.Print "After DoSub"
End Sub
```

Group MMP Flow Control Commands

9. Err Object

Summary Set Err to zero to clear the last error event. Err in an expression returns the last error code. Add vbObjectError to your error number in ActiveX Automation objects. Use Err.Raise or Error to trigger an error event. Err[.Number] This is the error code for the last error event. Set it to zero (or use Err.Clear) to clear the last error condition. Use Error or Err.Raise to trigger an error event. This is the default property. Err.Description This string is the description of the last error event. Err.Source This string is the error source file name of the last error event. Err.HelpFile This string is the help file name of the last error event. Err.HelpContext This number is the help context id of the last error event. Err.Clear Clear the last error event.

Declaration Err

Example

```
Sub Main
    On Error GoTo Problem
    Err = 1 ' set to error #1 (handler not triggered)
    Exit Sub

    Problem: ' error handler
    Error Err ' halt macro with message
End Sub
```

Group MMP Error Handling Commands

10. Exit Instruction

Summary The exit instruction causes the macro to continue with out doing some or all of the remaining instructions. Exit Description All Exit all macros. Do Exit the Do loop. For Exit the For of For Each loop.

Declaration Exit {All|Do|For|Function|Property|Sub|While}

Example

```
Sub Main
    L$ = InputBox$("Enter Do, For, While, Sub or All:")
    Debug.Print "Before DoSub"
    DoSub UCase$(L$)
    Debug.Print "After DoSub"
End Sub

Sub DoSub(L$)
    Do
        If L$ = "DO" Then Exit Do
        I = I+1
    Loop While I < 10
    If I = 0 Then Debug.Print "Do was entered"

    For I = 1 To 10
        If L$ = "FOR" Then Exit For
    Next I
    If I = 1 Then Debug.Print "For was entered"

    I = 10
    While I > 0
        If L$ = "WHILE" Then Exit While
        I = I-1
    Wend
    If I = 10 Then Debug.Print "While was entered"

    If L$ = "SUB" Then Exit Sub
```

```
    Debug.Print "Sub was not entered."
    If L$ = "ALL" Then Exit All
    Debug.Print "All was not entered."
End Sub
```

Group MMP Flow Control Commands

11. For Statement

Summary Execute statements while Num is in the range First to Last. Parameter Description Num This is the iteration variable. First Set Num to this value initially. Last Continue looping while Num is in the range. See Step below. Step If this numeric value is greater than zero then the for loop continues as long as Num is less than or equal to Last. If this numeric value is less than zero then the for loop continues as long as Num is greater than or equal to Last. If this is omitted then one is used.

Declaration For Num = First To Last [Step Inc] statements Next [Num]

Example

```
Sub Main
    For I = 1 To 2000 Step 100
        Debug.Print I;I+I;I*I
    Next I
End Sub
```

Group MMP Flow Control Commands

12. For Each Statement

Summary Execute statements for each item in items. Parameter Description var This is the iteration variable. items This is the collection of items to be done.

Declaration For Each var In items statements Next [var]

Example

```
Sub Main
    Dim Document As Object
    For Each Document In App.Documents
        Debug.Print Document.Title
```

```
    Next Document
End Sub
```

Group MMP Flow Control Commands

13. Function Definition

Summary User defined function. The function defines a set of statements to be executed when it is called. The values of the calling arglist are assigned to the params. Assigning to name[type] sets the value of the function result. Function defaults to Public if Private, Public or Friend are not is specified.

Declaration [| Private | Public | Friend] _ [Default] _ Function name[type][([param[, ...]])] [As type[()]] statements End Function

Example

```
Function Power(X,Y)
    P = 1
    For I = 1 To Y
        P = P*X
    Next I
    Power = P
End Function

Sub Main
    Debug.Print Power(2,8) ' 256
End Sub
```

Group MMP Declaration Commands

14. Goto Instruction

Summary Go to the label and continue execution from there. Only labels in the current user defined procedure are accessible.

Declaration GoTo label

Example

```
Sub Main
    X = 2
Loop:
    X = X*X
    If X < 100 Then GoTo Loop
    Debug.Print X ' 256
End Sub
```

Group MMP Flow Control Commands

15. If Statement

Summary

Declaration If condexpr Then [instruction] [Else instruction] -or- If condexpr Then statements [ElseIf condexpr Then statements]... [Else statements] End If -or- If TypeOf objexpr Is objtype Then ...

Example

```
Sub Main
    S = InputBox("Enter hello, goodbye, dinner or sleep:")
    S = UCase(S)
    If S = "HELLO" Then Debug.Print "come in"
    If S = "GOODBYE" Then Debug.Print "see you later"
    If S = "DINNER" Then
        Debug.Print "Please come in."
        Debug.Print "Dinner will be ready soon."
    ElseIf S = "SLEEP" Then
        Debug.Print "Sorry."
        Debug.Print "We are full for the night"
    End If
End Sub
```

Group MMP Flow ControForm 1: Single line if statement. Execute the instruction following the Then if condexpr is True. Otherwise, execute the instruction following the Else. The Else portion is optional. Commands

16. InputBox$ Function

Summary Display an input box where the user can enter a line of text. Pressing the OK button returns the string entered. Pressing the Cancel button returns a null string. Parameter Description Prompt$ Use this string value as the prompt in the input box. Title$ Use this string value as the title of the input box. If this is omitted then the input box does not have a title. Default$ Use this string value as the initial value in the input box. If this is omitted then the initial value is blank. XPos When the dialog is put up the left edge will be at this screen position. If this is omitted then the dialog will be centered. YPos When the dialog is put up the top edge will be at this screen position. If this is omitted then the dialog will be centered.

Declaration InputBox[$](Prompt$[, Title$][, Default$][, XPos, YPos])

Example

```
Sub Main
    L$ = InputBox$("Enter some text:", _
           "Input Box Example","asdf")
    Debug.Print L$
End Sub
Integer Data Type
Group: Data Type
Description:
A 16 bit integer value.
```

Group MMP User Input Commands

17. LBound Function

Summary Return the lowest index. Parameter Description arrayvar Return the lowest index for this array variable. dimension Return the lowest index for this dimension of arrayvar. If this is omitted then return the lowest index for the first dimension.

Declaration LBound(arrayvar[, dimension])

Example

```
Sub Main
    Dim A(-1 To 3,2 To 6)
    Debug.Print LBound(A)   '-1
    Debug.Print LBound(A,1) '-1
```

```
    Debug.Print LBound(A,2) ' 2
End Sub
```

Group MMP Variable Info Commands

18. Main Sub

Summary Form 1: Each macro must define Sub Main. A macro is a "program". Running a macro starts the Sub Main and continues to execute until the subroutine finishes.
Form 2: A code module may define a Private Sub Main. This Sub Main is the code module initialization subroutine. If Main is not defined then no special initialization occurs.

Declaration Sub Main() ... End Sub -or- Private Sub Main() ... End Sub

Example

```
Sub Main
    MsgBox "Please press OK button"
    If MsgBox("Please press OK button",vbOkCancel) = vbOK Then
        Debug.Print "OK was pressed"
    Else
    Debug.Print "Cancel was pressed"
    End If
End Sub
```

Group MMP Declaration Commands

19. MsgBox Instruction/Function

Summary Show a message box titled Title$. Type controls what the message box looks like (choose one value from each category). Use MsgBox( ) if you need to know what button was pressed. The result indicates which button was pressed. Result Value Button Pressed vbOK 1 OK button vbCancel 2 Cancel button vbAbort 3 Abort button vbRetry 4 Retry button vbIgnore 5 Ignore button vbYes 6 Yes button vbNo 7 No button Parameter Description Message$ This string value is the text that is shown in the message box. Type This numeric value controls the type of message box. Choose one value from each of the following tables. Title$ This string value is the title of the message box. Button Value Effect vbOkOnly 0 OK button vbOkCancel 1 OK and Cancel buttons vbAbortRetryIgnore 2 Abort, Retry, Ignore buttons vbYesNoCancel 3 Yes, No, Cancel buttons vbYesNo 4 Yes and No buttons vbRetryCancel 5 Retry and Cancel buttons Icon Value Effect 0 No icon vbCritical 16 Stop icon vbQuestion 32 Question icon vbExclamation 48 Attention icon vbInformation 64 Information icon Default Value Effect vbDefaultButton1 0 First button vbDefaultButton2 256 Second button vbDefaultButton3 512 Third button Mode Value Effect vbApplicationModal 0 Application modal vbSystemModal 4096 System modal vbMsgBoxSetForeground &h10000 System modal

Declaration MsgBox Message$[, Type][, Title$] -or- MsgBox(Message$[, Type][, Title$])

Example

```
Sub Main
MsgBox "Please press OK button"
If MsgBox("Please press OK button",vbOkCancel) = vbOK Then
Debug.Print "OK was pressed"
Else
Debug.Print "Cancel was pressed"
End If
End Sub
```

Group MMP User Input Commands

20. On Error Instruction

Summary

Declaration On Error GoTo 0 -or- On Error GoTo label -or- On Error Resume Next

Example

```
Sub Main
    On Error Resume Next
    Err.Raise 1
    Debug.Print "RESUMING, Err=";Err
    On Error GoTo X
    Err.Raise 1
    Exit Sub

X:  Debug.Print "Err=";Err
    Err.Clear
    Debug.Print "Err=";Err
    Resume Next
```

```
End Sub
```

Group MMP Error HandlinForm 1: Disable the error handler (default). Commands

21. Operators

Summary These operators are available for numbers n1 and n2 or strings s1 and s2. If any value in an expression is Null then the expression's value is Null. The order of operator evaluation is controlled by operator precedence. Operator Description - n1 Negate n1. n1 ^ n2 Raise n1 to the power of n2. n1 * n2 Multiply n1 by n2. n1 / n2 Divide n1 by n2. n1 \ n2 Divide the integer value of n1 by the integer value of n2. n1 Mod n2 Remainder of the integer value of n1 after dividing by the integer value of n2. n1 + n2 Add n1 to n2. s1 + s2 Concatenate s1 with s2. n1 - n2 Difference of n1 and n2. s1 & s2 Concatenate s1 with s2. n1 < n2 Return True if n1 is less than n2. n1 <= n2 Return True if n1 is less than or equal to n2. n1 > n2 Return True if n1 is greater than n2. n1 >= n2 Return True if n1 is greater than or equal to n2. n1 = n2 Return True if n1 is equal to n2. n1 <> n2 Return True if n1 is not equal to n2. s1 < s2 Return True if s1 is less than s2. s1 <= s2 Return True if s1 is less than or equal to s2. s1 > s2 Return True if s1 is greater than s2. s1 >= s2 Return True if s1 is greater than or equal to s2. s1 = s2 Return True if s1 is equal to s2. s1 <> s2 Return True if s1 is not equal to s2. Not n1 Bitwise invert the integer value of n1. Only Not True is False. n1 And n2 Bitwise and the integer value of n1 with the integer value n2. n1 Or n2 Bitwise or the integer value of n1 with the integer value n2. n1 Xor n2 Bitwise exclusive-or the integer value of n1 with the integer value n2. n1 Eqv n2 Bitwise equivalence the integer value of n1 with the integer value n2 (same as Not (n1 Xor n2)). n1 Imp n2 Bitwise implicate the integer value of n1 with the integer value n2 (same as (Not n1) Or n2).

Declaration ^ Not * / \ Mod + - & < <= > >= = <> Is And Or Xor Eqv Imp

Example

```
Sub Main
    N1 = 10
    N2 = 3
    S1$ = "asdfg"
    S2$ = "hjkl"
    Debug.Print -N1        '-10
    Debug.Print N1 ^ N2    ' 1000
    Debug.Print Not N1     '-11
    Debug.Print N1 * N2    ' 30
    Debug.Print N1 / N2    ' 3.3333333333333
    Debug.Print N1 \ N2    ' 3
    Debug.Print N1 Mod N2  ' 1
    Debug.Print N1 + N2    ' 13
```

```
    Debug.Print S1$ + S2$  '"asdfghjkl"
    Debug.Print N1 - N2    ' 7
    Debug.Print N1 & N2    '"103"
    Debug.Print N1 < N2    'False
    Debug.Print N1 <= N2   'False
    Debug.Print N1 > N2    'True
    Debug.Print N1 >= N2   'True
    Debug.Print N1 = N2    'False
    Debug.Print N1 <> N2   'True
    Debug.Print S1$ < S2$  'True
    Debug.Print S1$ <= S2$ 'True
    Debug.Print S1$ > S2$  'False
    Debug.Print S1$ >= S2$ 'False
    Debug.Print S1$ = S2$  'False
    Debug.Print S1$ <> S2$ 'True
    Debug.Print N1 And N2  ' 2
    Debug.Print N1 Or N2   ' 11
    Debug.Print N1 Xor N2  ' 9
    Debug.Print N1 Eqv N2  ' -10
    Debug.Print N1 Imp N2  ' -9
End Sub
```

Group MMP Operator Commands

22. Option Definition

Summary Require all variables to be declared prior to use. Variables are declared using Dim, Private, Public, Static or as a parameter of Sub, Function or Property blocks.

Declaration Option Explicit

Example

```
Option Explicit
```

```
Sub Main

    Dim A

    A = 1

    B = 2 ' B has not been declared

End Sub

Private Keyword

Group: Declaration

Description:

Private Consts, Declares, Functions, Propertys, Subs and Types
are only available in the current macro/module.

Public Keyword

Group: Declaration

Description:

Public Consts, Declares, Functions, Propertys, Subs and Types in
a module are available in all other macros/modules that access
it.
```

Group MMP Declaration Commands

23. ReDim Instruction

Summary Redimension a dynamic arrayvar or user defined type array element. Use Preserve to keep the array values. Otherwise, the array values will all be reset. When using preserve only the last index of the array may change, but the number of indexes may not. (A one-dimensional array can't be redimensioned as a two-dimensional array.)

Declaration ReDim [Preserve] name[type][([dim[, ...]])] [As type][, ...] -or- ReDim [Preserve] usertypevar.elem[type][([dim[, ...]])] [As type][, ...]

Example

```
Sub Main

    Dim X()

    ReDim X(3)

    Debug.Print UBound(X) ' 3

    ReDim X(200)
```

```
    Debug.Print UBound(X) ' 200
End Sub
```

Group MMP Declaration Commands

24. Select Case Statement

Summary Select the appropriate case by comparing the expr with each of the caseexprs. Select the Case Else part if no caseexpr matches. (If the Case Else is omitted then skip the entire Select...End Select block.) caseexpr Description expr Execute if equal. Is < expr Execute if less than. Is <= expr Execute if less than or equal to. Is > expr Execute if greater than. Is >= expr Execute if greater than or equal to. Is <> expr Execute if not equal to. expr1 To expr2 Execute if greater than or equal to expr1 and less than or equal to expr2.

Declaration Select Case expr [Case caseexpr[, ...] statements]... [Case Else statements] End Select

Example

```
Sub Main
    S = InputBox("Enter hello, goodbye, dinner or sleep:")
    Select Case UCase(S)
    Case "HELLO"
        Debug.Print "come in"
    Case "GOODBYE"
        Debug.Print "see you later"
    Case "DINNER"
        Debug.Print "Please come in."
        Debug.Print "Dinner will be ready soon."
    Case "SLEEP"
        Debug.Print "Sorry."
        Debug.Print "We are full for the night"
    Case Else
        Debug.Print "What?"
    End Select
End Sub
```

Group MMP Flow Control Commands

25. Set Instruction

Summary

Declaration Set objvar = objexpr -or- Set objvar = New objtype

Example

```
Sub Main
    Dim App As Object
    Set App = CreateObject("WinWrap.CppDemoApplication")
    App.Move 20,30 ' move icon to 20,30
    Set App = Nothing
    App.Quit       ' run-time error (no object)
End Sub
```

Group MMP AssignmenForm 1: Set objvar's object reference to the object reference of objexpr. Commands

26. Str$ Function

Summary Return the string representation of Num. Parameter Description Len Return the string representation of this numeric value. Positive values begin with a blank. Negative values begin with a dash '-'.

Declaration Str[$](Num)

Example

```
Sub Main
    Debug.Print Str$(9*9) ' 81
End Sub
String Data Type
Group: Data Type
Description:
An arbitrary length string value. Some useful string constants
are predefined:
 $(G    v (BbNullChar - same as Chr(0)
 $(G    v (BbCrLf - same as Chr(13) & Chr(10)
```

```
$(G    v (BbCr - same as Chr(13)
$(G    v (BbLf - same as Chr(10)
$(G    v (BbBack - same as Chr(8)
$(G    v (BbFormFeed - same as Chr(12)
$(G    v (BbTab - same as Chr(9)
$(G    v (BbVerticalTab - same as Chr(11)
```

Group MMP String Commands

27. Sub Definition

Summary User defined subroutine. The subroutine defines a set of statements to be executed when it is called. The values of the calling arglist are assigned to the params. A subroutine does not return a result. Sub defaults to Public if Private, Public or Friend are not is specified.

Declaration [| Private | Public | Friend] _ Sub name[([param[, ...]])] statements End Sub

Example

```
Sub IdentityArray(A()) ' A() is an array of numbers
    For I = LBound(A) To UBound(A)
        A(I) = I
    Next I
End Sub

Sub CalcArray(A(),B,C) ' A() is an array of numbers
    For I = LBound(A) To UBound(A)
        A(I) = A(I)*B+C
    Next I
End Sub

Sub ShowArray(A()) ' A() is an array of numbers
    For I = LBound(A) To UBound(A)
        Debug.Print "(";I;")=";A(I)
    Next I
```

```
End Sub


Sub Main
    Dim X(1 To 4)
    IdentityArray X() ' X(1)=1, X(2)=2, X(3)=3, X(4)=4
    CalcArray X(),2,3 ' X(1)=5, X(2)=7, X(3)=9, X(4)=11
    ShowArray X()     ' print X(1), X(2), X(3), X(4)
End Sub
```

Group MMP Declaration Commands

28. UBound Function

Summary Return the highest index. Parameter Description arrayvar Return the highest index for this array variable. dimension Return the highest index for this dimension of arrayvar. If this is omitted then return the highest index for the first dimension.

Declaration UBound(arrayvar[, dimension])

Example

```
Sub Main
    Dim A(3,6)
    Debug.Print UBound(A)   ' 3
    Debug.Print UBound(A,1) ' 3
    Debug.Print UBound(A,2) ' 6
End Sub
```

Group MMP Variable Info Commands

29. Wait Instruction

Summary Wait for Delay seconds.

Declaration Wait Delay

Example

```
Sub Main
```

```
    Wait 5 ' wait for 5 seconds
End Sub
```

Group MMP Miscellaneous Commands

30. While Statement

Summary Execute statements while condexpr is True.

Declaration While condexpr statements Wend

Example

```
Sub Main
    I = 2
    While I < 10
        I = I*2
    Wend
    Debug.Print I ' 16
End Sub
```

Group MMP Flow Control Commands

SLA Commands Reference

Following are the SLA Commands which can be used in a Multi Motor Program (MMP). They are arranged in related groups and in alphabetic order for easy access. All the SLA commands can be used in a MMP program by using "sla." namespace (type CTRL-SPACE after typing sla. in the MMP Programming environment.)

SLA Commands in Related Groups

The SLA Commands are categorized in each group depending on their functionalities.

- **SLA Initialization Commands**

  These commands are used in setting the initial environment before a program can be successfully started. They are usually called in the initialization method before any other command is executed.

  - Calibrate
  - Reset

- **SLA Motion Commands**

  These commands are used to move the slides to specified positions or along the specified paths. They also contain commands to obtain the useful information regarding positions to move to.

  - BeginningPathPoint
  - CurrentPoint
  - DoLine
  - DoPath
  - DoPositionMove
  - DoRelativeMove
  - DoVelocityMove
  - MaxOfPath
  - MinOfPath
  - StopMotion
  - WaitForStop

- **SLA I/O Commands**

  These commands are used to control the I/O switches. They can be used to control the integrated I/O as well as externally supplied unit.

  - CheckSwitch
  - SetSwitch
  - WaitForSwitch

- ## SLA Motor Commands

  These commands directly related to individual motors that drive the slides. They can be used to obtain useful status information regarding motors.

  - MotorStatus
  - RunDiagnostics
  - SetServoOff

- ## SLA Multi Tasking Commands

  These commands can be used to launch multiple programs simultaneously. These programs run at the same time and can be very useful where multi-tasking capability is needed.

  - StartProgram
  - StopProgram

- ## SLA State Management Commands

  These commands can be used to maintain the custom state information in the shared memory space which can be optionally made persistent (saved on the disk). The shared memory can be used by multiple tasks to communicate among themselves.

  - GetValue
  - RemoveValue
  - SaveValue

- ## SLA Axis Transformation Commands

  These commands can be used for axis transformations of the existing position data for reusing in different reference frames. These commands have the chaining capability to combine multiple transformations for very powerful results.

  - AppendPaths
  - ApplyTransformation
  - ApplyTransformationToDataset
  - ApplyTransformationToPath
  - ApplyTransformationToPoint
  - CurrentCoordinateSystem
  - ReflectAxis
  - ReversePath
  - RotateAxis
  - RotateAxisAroundPoint
  - ScaleAxis

- ShiftAxis

- **SLA Instrumentation Commands**

  These commands can be used to measure time profile behavior of a program. They have capabilities to profile multiple sections of the running program at the same time.

  - ElapsedTimer
  - StartTimer

- **SLA Miscellaneous Commands**

  These commands provide additional capabilities to augment the existing powerful command set.

  - LogMessage
  - SendMail

SLA Commands in Alphabetic Order

1. AppendPaths
2. ApplyTransformation
3. ApplyTransformationToDataset
4. ApplyTransformationToPath
5. ApplyTransformationToPoint
6. BeginningPathPoint
7. Calibrate
8. CheckSwitch
9. CurrentCoordinateSystem
10. CurrentPoint
11. DoLine
12. DoPath
13. DoPositionMove
14. DoRelativeMove
15. DoVelocityMove
16. ElapsedTimer
17. GetValue
18. LogMessage
19. MaxOfPath
20. MinOfPath
21. MotorStatus
22. ReflectAxis
23. RemoveValue
24. Reset
25. ReversePath
26. RotateAxis

SLA Commands

[note: *In the example snippets shown below, MyDataset is the name of a dataset created by user. path1, path2 etc. are the names of the paths in the MyDataset which can be accessed using the fully qualified names MyDataset.path1, MyDataset.path2 etc. And, point1, point2 etc. are the points in the MyDataset which can be accessed using the fully qualified name MyDataset.point1, MyDataset.point2 etc.*]

1. AppendPaths

Summary Append the paths defined in a dataset to create a concatenated path.

Declaration AppendPaths(first As Path, Optional second As Path)

Return Returns appended path.

Description AppendPaths command appends second path to the first path and returns reference to first path. If optional second path is not specified, it removes all appended paths from the first path if any. Since it returns the first path, the return value can be used to append subsequent paths as shown in the following example.

Example Following example appends path2 and path3 to path1 and returns path1. Note that it is necessary to remove any earlier appended paths from path1 to prevent continuous chaining of the paths from the earlier calls.

```
                sla.AppendPaths(MyDataset.path1, Nothing) 'remove
any earlier appended paths

                sla.AppendPaths(sla.AppendPaths(MyDataset.path1,
MyDataset.path2), MyDataset.path3)

                sla.DoPath (MyDataset.path1, 100, 1000, 5)
```

Group SLA Axis Transformation Commands

2. ApplyTransformation

Summary Apply transformation to all position data.

Declaration ApplyTransformation ()

Return No return value.

Description ApplyTransformation command applies the transformations to all the datasets. This could potentially be expensive operation and it is recommended that the more specific commands ApplyTransformationToPoint, ApplyTransformationToPath or ApplyTransformationToDataset be used depending on the position data of interest. Until any of the ApplyTransformation* commands are applied, the transformations do not affect any of the position data.

Example Following example applies transformation to all the paths and points (in all datasets).

```
sla.ApplyTransformation()
```

Group SLA Axis Transformation Commands

3. ApplyTransformationToDataset

Summary Apply transformation to the specified dataset.

Declaration ApplyTransformationToDataset(datasetToTransform As DataSet)

Return No return value.

Description ApplyTransformationToDataset command applies the transformation to all the paths and points contained in the specified dataset. If only few paths or points out of all the existing paths and points in a dataset are required to transform, consider using the more specific ApplyTransformationToPath or ApplyTransformationToPoint commands to reduce the transformation time.

Example Following example applies transformation to all the paths and points in MyDataset.

```
sla.ApplyTransformationToDataset(MyDataset)
```

Group SLA Axis Transformation Commands

4. ApplyTransformationToPath

Summary Apply transformation to the specified path.

Declaration ApplyTransformationToPath(pathToTransform As Path)

Return No return value.

Description ApplyTransformationToPath command applies the transformation to the specified path.

Example Following example applies transformation to the specified MyDataset.path1.

```
sla.ApplyTransformationToPath(MyDataset.path1)
```

Group SLA Axis Transformation Commands

5. ApplyTransformationToPoint

Summary Apply transformation to the specified point.

Declaration ApplyTransformationToPoint(pointToTransform As Point)

Return No return value.

Description ApplyTransformationToPoint command applies the transformation to the specified point.

Example Following example applies transformation to the specified MyDataset.point1.

```
sla.ApplyTransformationToPoint(MyDataset.point1)
```

Group SLA Axis Transformation Commands

6. BeginningPathPoint

Summary Return the beginning point of the specified path.

Declaration BeginningPathPoint(specifiedPath As Path)

Return Returns the beginning point.

Description BeginningPathPoint command returns the first point of a path which can be used to move to the beginning of the path. If the path is linear then it returns the first point on the path, while for an arc, it returns the beginning point of the arc. The returned point can be stored in a Point type variable and can be subsequently used wherever a command takes Point as an argument e.g. DoLine.

Example Following example returns the first point of MyDataset.path1..

```
Dim beginningPoint as Point

Set beginningPoint =
sla.BeginningPathPoint(MyDataset.path1)
```

Group SLA Motion Commands

7. Calibrate

Summary Calibrate the motors by homing them in the desired direction.

Declaration Calibrate(Optional motorIndex As Integer, Optional direction As Integer, Optional force As Boolean, Optional speed As Double, Optional acceleration As Double)

Return No return value.

Description Calibrate command homes a motor in either direction (-1 for negative direction towards the motor (default), +1 for positive direction away from the Motor). Once calibrated, a motor won't home again unless the optional force parameter is specified as True. motorIndex parameter can be 0 for homing all motors, or a particular motor (1, 2 or 3) can be specified for homing only one motor. If no values are specified, the command will home all motors in the negative direction only if they haven't been homed earlier. The optional speed and acceleration parameters can be used to control the speed of homing.

Example Following example performs homing of motorX in positive direction irrespective of whether it was done earlier.

```
sla.Calibrate(1, 1, True)
```

Group SLA Initialization Commands

8. CheckSwitch

Summary Check the status of an inputSwitch or outputSwitch.

Declaration CheckSwitch(motorIndex As Integer, switchName As String, Optional isInputSwitch As Boolean)

Return Return the state (0 for off, 1 for on) of the switch.

Description CheckSwitch command gets the status of the switch on the indicated IO unit. The optional parameter isInputSwitch (True by default) determines whether it is input or output switch. The motorIndex parameter could be any of the motors (1, 2 or 3) or external IO indicated by 0. The switch names can be defined on the configuration screen. The returned value is either 0 (for switch in low/off status) or 1 (for switch in high/on status).

Example Following examples show how to obtain the switch status for different devices. In the first example the command retrieves the value of "MotorX_Input_Switch" located on motorX. In the second example the command retrieves the value of "External_IO_Output_Switch" located on external IO unit (optional).

```
Dim motorInputSwitchStatus as Integer

motorInputSwitchStatus = sla.CheckSwitch (1,
"MotorX_Input_Switch")
```

```
Dim externalOutputSwitchStatus as Integer

externalOutputSwitchStatus = sla.CheckSwitch (0,
"External_IO_Input_Switch", False)
```

Group SLA I/O Commands

9. CurrentCoordinateSystem

Summary Set the current coordinate system.

Declaration CurrentCoordinateSystem (systemName As String)

Return Returns True or False depending on the success or failure of the command.

Description CurrentCoordinateSystem command changes the current coordinate system. Valid values are "WORLD" or "USER". Set coordinate system to "WORLD" to use the original (untransformed) data points. Set coordinate system to "USER" to use the transformed data points.

Example Following example changes the coordinate system to "USER"

```
sla.CurrentCoordinateSystem ("USER")
```

Group SLA Axis Transformation Commands

10. CurrentPoint

Summary Return the current location.

Declaration CurrentPoint()

Return Returns a point representing current location.

Description CurrentPoint command returns the current location. Individual axis values can be obtained by accessing the members of the returned point.

Example Following example stores the current value of motorX in currentX variable.

```
Dim currentX as Double
currentX = sla.CurrentPoint().x
```

Group SLA Motion Commands

11. DoLine

Summary Move a point in linear trajectory using coordinated motion.

Declaration DoLine(endPoint As Point, speed As Double, acceleration As Double)

Return No return value.

Description DoLine command moves the current point in linear motion to the specified destination with given speed and acceleration. The command blocks the program execution till the motion is completed.

Example Following example will move the slides to MyDataset.point1 from current location with speed 100 and acceleration 1000

```
sla.DoLine (MyDataset.point1, 100, 1000)
```

Group SLA Motion Commands

12. DoPath

Summary Move a point along a user defined path using coordinated motion.

Declaration DoPath(pathToMoveAlong As Path, speed As Double, acceleration As Double, Optional cornerRadius As Double)

Return No return value.

Description DoPath command moves the current point along the user defined path with given speed and acceleration. A path can be a linear path (made up of connected straight lines) or an arc path (arc of a circle). For a linear path, the optional cornerRadius argument can be used to specify any desired rounding of the corners. The corners where the rounding can not be achieved, the cornerRadius argument will have no effect. The rounding of the corners helps with a smoother trajectory, thus helping to reduce any vibration which can be present due to sudden change in direction. This rounding works in all geometries including three dimensional paths. The command blocks the program execution till the motion is completed.

Example Following example moves the point along the MyDataset.path1 with speed 100 and acceleration 1000 rounding the corners with arc of 5 units wherever possible.

```
sla.DoPath (MyDataset.myPath, 100, 1000, 5)
```

Group SLA Motion Commands

13. DoPositionMove

Summary Do position move to the specified location.

Declaration DoPositionMove(motorIndex As Integer, absolutePosition As Double, speed As Double, acceleration As Double)

Return No return value.

Description DoPositionMove command moves a motor to the specified absolutePosition with given speed and acceleration for the motor indicated by motorIndex. Valid values of motorIndex are 1 for motorX, 2 for motorY and 3 for motorZ. This is a non-blocking command and the program execution will continue even while the motion is in progress. This is useful in optimizing cycle times by not waiting for a motor to stop. To wait till the motion is complete, use it in combination with WaitForStop command.

Example Following example moves motorX to position 50 with speed 100 and acceleration 1000.

```
sla.DoPositionMove(1, 50, 100, 1000)
```

Group SLA Motion Commands

14. DoRelativeMove

Summary Do relative move from the current position in given direction.

Declaration DoRelativeMove(motorIndex As Integer, relativeMoveDistance As Double, speed As Double, acceleration As Double)

Return No return value.

Description DoRelativeMove command moves a motor in either direction by specified relativeMoveDistance (positive value moves in positive direction, negative value moves in negative direction) with given speed and acceleration for the motor indicated by motorIndex. Valid values of motorIndex are 1 for motorX, 2 for motorY and 3 for motorZ. This is a non-blocking command and the program execution will continue even while the motion is in progress. This is useful in optimizing cycle times by not waiting for a motor to stop. To wait till the motion is complete, use it in combination with WaitForStop command.

Example Following example moves motorX in positive direction by 50 with speed 100 and acceleration 1000.

```
sla.DoRelativeMove(1, 50, 100, 1000)
```

Group SLA Motion Commands

15. DoVelocityMove

Summary Do constant velocity move for a motor in given direction.

Declaration DoVelocityMove(motorIndex As Integer, direction As Integer, speed As Double, acceleration As Double)

Return No return value.

Description DoVelocityMove command moves a motor in either direction (+1 for positive direction, -1 for negative direction) with given speed and acceleration for the motor indicated by motorIndex. Valid values of motorIndex are 1 for MotorX, 2 for MotorY and 3 for MotorZ. This is a non-blocking command and the program execution will continue even while the motion is in progress. This is useful in optimizing cycle times by not waiting for a motor to stop. To wait till the motion is complete, use it in combination with WaitForStop command.

Example Following examples moves the motorX in negative direction with speed 100 and acceleration 1000 till it stops (hits the limit switch).

```
sla.DoVelocityMove(1, -1, 100, 1000)
```

Group SLA Motion Commands

16. ElapsedTimer

Summary Return the elapsed time for the timer.

Declaration ElapsedTimer(timerKey As String, Optional resetTimer As Boolean)

Return Returns elapsed time in seconds since the start of the timer.

Description ElapsedTimer command returns the time in seconds since the start of the timer. First the timer with the same timerKey has to be started using StartTimer command for this command to work. If optional resetTimer argument is specified to be True (default is False), then the timer will reset it. This can be useful while using the same timer for multiple measurements.

Example Following example assigns the myTimer's current value to elapsedTime variable and resetting the timer.

```
Dim elapsedTime as Double

elapsedTime = sla.ElapsedTimer("myTimer", True)
```

Group SLA Instrumentation Commands

17. GetValue

Summary Get a stored state using the key.

Declaration GetValue(key As String)

Return Returns stored value associated with the key as String.

Description GetValue command retrieves the earlier stored value using the specified key. The returned value can be cast back to its original type using appropriate CDbl(), CInt(), CLng(), CSng() or CBool() functions which take a String as the argument. If no stored values are found for the specified key, an empty string is returned.

Example Following is an example of getting a stored Integer value with "result" as the key.

```
Dim value as Integer

value = CInt(sla.GetValue("result"))
```

Group SLA State Management Commands

18. LogMessage

Summary Log a message with given log level.

Declaration LogMessage (message As String, Optional level As Integer)

Return No return value.

Description Logs a message to log window as well as logs.txt file located by default at C:\Program Files\Robohand\SLA\logs\logs.txt with the given log level. Depending on the log level value, the message is highlighted in different colors in the log window. The valid log level values are 1 (information), 2 (warning) or 3 (error).

Example Following is an example for logging message at information level.

```
sla.LogMessage("This is an example log message
for information", 1)
```

Group SLA Miscellaneous Commands

19. MaxOfPath

Summary Get the maximum value of specified axis for the path.

Declaration MaxOfPath(path As Path, axis As Integer)

Return Returns maximum value of specified axis for any point in the path as Double.

Description MaxOfPath command gets the maximum value of specified axis (1, 2, or 3) for the path. This is useful in defining the envelope of operation.

Example Following is an example of getting maximum Y axis value in MyDataset.path1 and assigning it to maxY variable.

```
Dim maxY as Double

maxY = sla.MaxOfPath(MyDataset.path1, 2)
```

Group SLA Motion Commands

20. MinOfPath

Summary Get the minimum value of specified axis for the path.

Declaration MinOfPath (path As Path, axis As Integer)

Return Returns minimum value of specified axis for any point in the path as Double.

Description MinOfPath command gets the minimum value of specified axis (1, 2, or 3) for the path. This is useful in defining the envelope of operation.

Example Following is an example of getting minimum X axis value in MyDataset.path1 and assigning it to minX variable.

```
Dim minX as Double

minX = sla.MinOfPath(MyDataset.path1, 1)
```

Group SLA Motion Commands

21. MotorStatus

Summary Get and set the motor status information.

Declaration MotorStatus(motorIndex As Integer)

Return Returns an object of MotorStatus type.

Description MotorStatus command returns an object of MotorStatus type which encompasses the relevant status flags information for the specified motorIndex. Valid values of motorIndex are 1 for MotorX, 2 for MotorY and 3 for MotorZ. Individual status flag value can be obtained by accessing the members of the returned object. Currently available status flags are as follows and meaning of the flags when they are True.

Ba - over current state
Bd - user math overflow
Be - excessive position error
Bh - excessive temperature (real time)
Bk - over current state (real time)
Bm - left limit (real time)
Bo - motor off (real time)
Bp - right limit (real time)
Bs - syntax error
Bt - trajectory in progress (real time)
Bu - user array index range error

Apart from these statuses, additional motor status information available is:

MaxCurrent - maximum current limit (0 to 1023, default = 1000). In some applications, if the motor is misapplied full power, the attached mechanism could be damaged. It can be useful to reduce the maximum amount of current available thus limiting the torque the motor can put out.

MaxPositionError - maximum position error (1 to 32000, default = 1000). The difference between where the motor shaft is and where it is supposed to be is appropriately called the 'error'. The magnitude and sign of the error is delivered to the motor in the form of torque, after it is put through the PID filter. The higher the error, the more out of control the motor is. Therefore, it is often useful to put a limit on the allowable error, after which time the motor will be turned off.

Calibration - It returns true/false depending on whether the motor has been calibrated or not. This information can be used to alert an operator before proceeding with the calibration.

Example Following example retrieves value of Bt (whether motor trajectory in progress) for MotorX and stores it in isMoving Boolean variable. The second example sets the maximum position error 500 on MotorX which will trigger position error at much tighter variations.

```
Dim isMoving as Boolean

isMoving = sla.MotorStatus(1).Bt


sla.MotorStatus(1).MaxPositionError = 500
```

Group SLA Motor Commands

22. ReflectAxis

Summary Reflect specified axis.

Declaration ReflectAxis(Optional xReflection As Boolean, Optional yReflection As Boolean, Optional zReflection As Boolean , Optional isChained As Boolean)

Return No return value.

Description ReflectAxis command multiplies -1 to the x, y or z axis values in the WORLD coordinate system to create transformed values in the USER coordinate system. Since each of the reflection is optional, it is possible to perform reflection only in the desired coordinate. Using the optional isChained argument (False by default), the reflect transformation can be chained with any earlier defined transformation to create a more complex combined transformation. A transformation has to be explicitly applied to the position data using one of the ApplyTransformation* command to have any effect on the motion. It is useful to note that the same effect can be achieved by using ScaleAxis command with scaleFactor of -1.

Example Following example shows how to reflect y axis values and remove any of the earlier transformations by specifying isChained as False. The equivalent command using ScaleAxis is also shown below.

```
sla.ReflectAxis (False, True, False, False)

sla.ScaleAxis(1, -1, 1, False)
```

Group SLA Axis Transformation Commands

23. RemoveValue

Summary Remove a state from memory and persistent storage (if optionally stored).

Declaration RemoveValue(key As String)

Return No return value.

Description RemoveValue command removes an earlier stored value associated with the key. If the value was also stored in persistent storage, that value is removed as well. If the value is not removed it continues to be available to all the programs till the SLA Operating Software terminates. It is recommended that any unused persistent values be removed to save disk space.

Example Following is an example of removing a stored value with "result" as the key.

```
sla.RemoveValue("result")
```

Group SLA State Management Commands

24. Reset

Summary Reset the SLA Operating Software.

Declaration Reset()

Return No return value.

Description Reset command does multiple reset operations. It resets the coordinate system back to "WORLD", removes any transformation information, removes the internal cache used to speed up the coordinated motion moves, clears all the status flags from motors, and removes any appended paths as a result of AppendPaths command. Reset command should be used only once at the beginning of the program. The MMP program template used to create a new MMP program automatically includes call to Reset command in its initialization method.

Example Following is an example of resetting the SLA Operating Software.

```
sla.Reset()
```

Group SLA Initialization Commands

25. ReversePath

Summary Reverse the indicated axis values.

Declaration ReversePath(Optional xReverse As Boolean, Optional yReverse As Boolean, Optional zReverse As Boolean)

Return No return value.

Description ReversePath command reverses the order of values in the indicated coordinate for a path. By default the command will reverse all the three coordinates, effectively reversing the path (the last point of the path becomes the first point and vice versa). If only a specific coordinate needs to be reversed, pass False to other coordinates. This along with ReflectAxis can be used to create a continuous path combining the original and the reflected path. This command differs from other transformation command since it specifically applies only to a path and can not be chained. A transformation has to be explicitly applied to the position data using one of the ApplyTransformation* command to have any effect on the motion.

Example Following examples show how to reverse a whole path or a specific coordinate. The first example reverses the whole path and the second example reverses only the x coordinates of the points in a path.

```
sla.ReversePath()

sla.ReversePath( , False, False)
```

Group SLA Axis Transformation Commands

26. RotateAxis

Summary Rotate around specified axis by given angle (degrees).

Declaration RotateAxis (Optional angle As Double, Optional aroundAxis As Integer, Optional isChained As Boolean)

Return No return value.

Description RotateAxis command rotates a point in orthogonal plane around the specified axis (1 - default, 2 or 3) in the WORLD coordinate system to create transformed values in the USER coordinate system. Using the optional isChained argument (False by default), the rotate transformation can be chained with any earlier defined transformation to create a more complex combined transformation. A transformation has to be explicitly applied to the position data using one of the ApplyTransformation* command to have any effect on the motion. It is useful to note that the same effect can be achieved by using RotateAxisAroundPoint command with origin as aroundPoint parameter.

Example Following example shows how to rotate a point in x-y plane by 90 degrees and remove any of the earlier transformations by specifying isChained as False.

```
sla.RotateAxis(90, 3, False)
```

Group SLA Axis Transformation Commands

27. RotateAxisAroundPoint

Summary Rotate around specified point by given angle (degrees).

Declaration RotateAxisAroundPoint(Optional aroundPoint As Point, Optional angle As Double, Optional aroundAxis As Integer, Optional isChained As Boolean)

Return No return value.

Description RotateAxisAroundPoint command rotates a point in orthogonal plane around the axis that passes thorough the specified point and is parallel to the aroundAxis (1 - default, 2 or 3) in the WORLD coordinate system to create transformed values in the USER coordinate system. Using the optional isChained argument (False by default), the rotate transformation can be chained with any earlier defined transformation to create a more complex combined transformation. A transformation has to be explicitly applied to the position data using one of the ApplyTransformation* command to have any effect on the motion.

Example Following example shows how to rotate a point in x-y plane around MyDataset.point1 by 90 degrees and remove any of the earlier transformations by specifying isChained as False.

```
sla.RotateAxisAroundPoint(MyDataset.point1, 90,
3, False)
```

Group SLA Axis Transformation Commands

28. RunDiagnostics

Summary Run diagnostics to recover from unexpected motor problems.

Declaration RunDiagnostics()

Return Returns True or False depending on the success or failure of the command.

Description RunDiagnostics runs several internal diagnostics to fix a problem which can't be fixed by user level commands. For example, when power is turned off and on again (e.g. in the event of an E-Stop) while the SLA OS program is running, the serial connection becomes invalid. In this case no communication can take place between the software and the motors. Running this command reconnects to the motors again with the new connections and allowing the communication to take place.

Example Following is an example of running diagnostics to fix unexpected problems.

```
sla.RunDiagnostics()
```

Group SLA Motor Commands

29. SaveValue

Summary Save a state in memory or persistent storage for later retrieval.

Declaration SaveValue(key As String, value As String, Optional isPersistence As Boolean)

Return No return value.

Description SaveValue command stores a String value in program memory or optionally on the disk when isPersistence is declared True (False by default). When a value is stored in memory, it is not available when the SLA Operating Software exits e.g. in case of power failure. If there is some critical information that needs to survive the unexpected exit of the SLA Operating Software, it must be stored with isPersistence flag to be True. Since all programs share the same memory space, it is important to choose the name of keys judiciously to avoid overwriting. Another very important usage is as a shared memory space for communicating among different programs when Multitasking feature is used. A program running in one thread can SaveValue, which can be retrieved by another program running in different thread using GetValue. Since both key and value arguments are passed as String, any numerical value must be converted to String using CStr() function.

Example Following is an example of storing a Integer value using "result" as the key.

```
Dim value as Integer

value = 10

sla.SaveValue("result", CStr(value))
```

Group SLA State Management Commands

30. ScaleAxis

Summary Scale axis by specified scaleFactor.

Declaration ScaleAxis(Optional xScale As Double, Optional yScale As Double, Optional zScale As Double, Optional isChained As Boolean)

Return No return value.

Description ScaleAxis command multiplies the scaleFactor (either negative or positive) specified by xScale, yScale and zScale to the x, y or z axis values in the WORLD coordinate system to create transformed values in the USER coordinate system. Since each of the scaleFactor is optional, it is possible to perform scaling only in the desired coordinate. Using the optional isChained argument (False by default), the scale transformation can be chained with any earlier defined transformation to create a more complex combined transformation. A transformation has to be explicitly applied to the position data using one of the ApplyTransformation* command to have any effect on the motion.

Example Following example shows how to multiply x and z axis values by 2 and remove any of the earlier transformations by specifying isChained as False.

```
sla.ScaleAxis(2, ,2, False)
```

Group SLA Axis Transformation Commands

31. SendMail

Summary Send email to the specified address.

Declaration SendMail(smtpServer As String, recipient As String, message As String)

Return No return value.

Description SendMail command sends an email containing the message to the specified recipient using the smtpServer for relay. The sender will be sla@robohandsla.com (SLA Software).

Example Following is an example of sending email to Quality Assurance department of an example company.

```
sla.SendMail("smtp.exampleCompany.com",
"qa@exampleCompany.com", "SLA Operating Software completed
successfully!")
```

Group SLA Miscellaneous Commands

32. SetServoOff

Summary Turn off servo in the motor, optionally turning off the automatic brakes.

Declaration SetServoOff (motorIndex As Integer, Optional autoBrake As Boolean)

Return No return value.

Description SetServoOff command turns off servo in the indicated motor (1, 2 or 3). If there is a need for moving motors around by hand, the automatic brakes (on by default) can be turned off by supplying the optional argument autoBrake as False. Since turning the servo off and the brakes at the same time can lead to undesired motions (e.g. z slide falling down due to gravity), exercise caution in using the command.

Example Following examples turns off servo of motorX, while maintaining the brakes.

```
sla.SetServoOff(1, True)
```

Group SLA Motor Commands

33. SetSwitch

Summary Set the outputSwitch to specified status.

Declaration SetSwitch (motorIndex As Integer, outputSwitch As String, status As Integer)

Return No return value.

Description SetSwitch command sets outputSwitch on the indicated IO unit to specified status. The motorIndex parameter could be any of the motors (1, 2 or 3) or external IO indicated by 0. The outputSwitch names can be defined on the configuration screen. The status could be 0 (for setting outputSwitch to low/off) or 1 (for setting outputSwitch to high/on).

Example Following examples show how to set the output switch status for different devices. In the first example the command sets the value of "MotorX_Output_Switch" located on motorX to 1. In the second example the command sets the value of "External_IO_Output_Switch" located on external IO unit (optional) to 0.

```
sla.SetSwitch (1, "MotorX_Output_Switch", 1)

sla.SetSwitch (0, "External_IO_Output_Switch", 0)
```

Group SLA I/O Commands

34. ShiftAxis

Summary Shift axis by specified distances.

Declaration ShiftAxis(Optional xShift As Double , Optional yShift As Double, Optional zShift As Double, Optional isChained As Boolean)

Return No return value.

Description ShiftAxis command adds the shiftDistance (either negative or positive) specified by xShift, yShift and zShift to the x, y or z axis values in the WORLD coordinate system to create transformed values in the USER coordinate system. Since each of the shiftDistance is optional, it is possible to perform shifts only in the desired coordinate. Using the optional isChained argument (False by default), the shift transformation can be chained with any earlier defined transformation

to create a more complex combined transformation. A transformation has to be explicitly applied to the position data using one of the ApplyTransformation* command to have any effect on the motion.

Example Following example shows how to shift y axis values by 20 units and remove any of the earlier transformations by specifying isChained as False.

```
sla.ShiftAxis( , 20, , False)
```

Group SLA Axis Transformation Commands

35. StartProgram

Summary Start another program in a different thread.

Declaration StartProgram(programName As String)

Return No return value.

Description StartProgram command spawns the specified program in a new thread and begins its execution. Since this is a non-blocking command, the original program continues to run at the same time. Once started, both the programs run independent of each other i.e. no parent-child relationship is maintained. These programs can communicate among themselves using the shared memory provided by SLA State Management Commands. Running multiple programs simultaneously provides a very powerful multi-tasking capability to SLA Operating Software. However, since these threads are sharing the same CPU, the available cycle time gets divided among them which may result in slowing down of individual programs. In the time sensitive operations, it is recommended to run only one program at a time. Also make sure that only one of the concurrently running program contains the motion related commands, as otherwise it can result in an unpredictable behavior. Note that it is an idempotent command i.e. multiple invocations for starting the same program will result in only one instance of the program starting and all subsequent requests will be ignored. Typical application includes running the motors in one program while monitoring inputs or controlling outputs in another program.

Example Following example starts another program "program2.mmp".

```
sla.StartProgram("program2.mmp")
```

Group SLA Multi Tasking Commands

36. StartTimer

Summary Start the specified timer.

Declaration StartTimer(timerKey As String)

Return No return value.

Description StartTimer command starts a timer with the specified timerKey. Multiple timers can be started by specifying different timerKeys. The command ElapsedTimer is used to retrieve the current value of the timer using the timerKey.

Example Following example starts the myTimer.

```
sla.StartTimer("myTimer")
```

Group SLA Instrumentation Commands

37. StopMotion

Summary Stop motion of the specified motor.

Declaration StopMotion(Optional motorIndex As Integer, Optional deaccelerate As Boolean)

Return No return value.

Description StopMotion command terminates the current trajectory of the specified motor. The motorIndex parameter could be any of the motors (1, 2 or 3). If no argument is specified, all motors (default) are stopped. The second optional parameter deaccelerate (True by default), controls the velocity profile while stopping the motors. If True then the motors deaccelerate to stop, otherwise they stop immediately. If the command StopMotion is used with the coordinated motion commands DoLine or DoPath from a different thread, it can result in an error.

Example Following example deaccelerates motorX to stop.

```
sla.StopMotion(1, True)
```

Group SLA Motion Commands

38. StopProgram

Summary Stop another program running in the different thread.

Declaration StopProgram(programName As String)

Return No return value.

Description StopProgram command terminates the specified program running in the different thread. Since all the programs run independent of each other i.e. no parent-child relationship is maintained, stopping of a parent program has no effect on child program or vice versa. The program terminates after completing the currently executing statement.

Example Following example stops another program "program2.mmp".

```
sla.StopProgram("program2.mmp")
```

Group SLA Multi Tasking Commands

39. WaitForStop

Summary Wait for the motion to be completed.

Declaration WaitForStop(Optional motorIndex As Integer, Optional monitorMotorIndex As Integer, Optional monitorInputSwitch As String, Optional monitorInputSwitchStatus As Integer, Optional decelerate As Boolean, Optional timeOutSeconds As Long)

Return Returns True if all motors did stop or False otherwise (including return due to timeOut)

Description WaitForStop command waits for the end of motion of all (default) motors. If the optional motorIndex argument is specified then it waits for the end of motion only for that motor. The motorIndex parameter could be any of the motors (1, 2 or 3). The subsequent three optional arguments can be used together to provide a monitor signal which when activated (has the value specified by monitorInputSwitchStatus) stops the motion. The values for monitorMotorIndex are the same as that for CheckSwitch command (1,2,3 for IO's on motors or 0 for external IO). monitorInputSwitch is the name of the input switch to monitor and monitorInputSwitchStatus indicates on what status (0 or 1) the motion stops. The optional parameter decelerate (True by default), controls the velocity profile while stopping the motor. If True then the motor decelerates to stop, otherwise it stops immediately. The optional parameter timeOutSeconds allows it to wait only for specified seconds. If the motors are still in motion after timeOutSeconds, command will return with the False return value. If a motor stops due to fault (e.g. position error or limit switch), then an error condition is raised which will usually end the program. This command blocks the program execution till either the motion stops (if no monitor related or timeOut arguments are specified) or when the monitor signal is activated (if monitor related arguments are specified) or when the timeOut period is expired. This command can be used along with DoRelativeMove, DoPositionMove or DoVelocityMove commands to achieve synchronized motion (as opposed to coordinated motion where each of the motors move in lockstep manner) where each motor runs independently to obtain highly optimized timing profile.

Example Following examples demonstrate the command with and without the external signal monitoring. The first example shows starting synchronized motion by using DoRelativeMove and waiting for completion of motion of all motors with timeOut parameter of 5 seconds. MotorX is doing relative move from the current location for 100 units with speed 100 and acceleration 1000. MotorY is doing relative move from the current location for 200 units with speed 200 and acceleration 1000. MotorX is doing relative move from the current location for 500 units with speed 500 and acceleration 1000. The use of WaitForStop ensures that the subsequent commands will execute after all motors have reached their destinations or five seconds have elapsed, whichever occurs first. The second example shows starting synchronized motion by using DoPositionMove and blocking till the "MotorX_Input_Switch" switch located on motorX goes high (1) or the motion is completed on all motors. If the "MotorX_Input_Switch" is activated before the motion ends, the motor will come to abrupt and immediate stop, since the parameter decelerate is False.

```
sla.DoRelativeMove 1, 100, 100, 1000
sla.DoRelativeMove 2, 200, 200, 1000
sla.DoRelativeMove 3, 500, 500, 1000
sla.WaitForStop(, , , , , 5)

sla.DoPositionMove 1, 100, 100, 1000
sla.DoPositionMove 2, 200, 200, 1000
sla.DoPositionMove 3, 500, 500, 1000
```

```
                sla.WaitForStop(0, 1, "MotorX_Input_Switch", 1,
False)
```

Group SLA Motion Commands

40. WaitForSwitch

Summary Wait for an input switch to attain specified status.

Declaration WaitForSwitch (motorIndex As Integer, inputSwitch As String, status As Integer, Optional timeOutSeconds As Long)

Return Returns True or False depending on the success or failure of the command.

Description WaitForSwitch command waits timeOutSeconds for an input switch to attain the specified status on the indicated IO unit. The motorIndex parameter could be any of the motors (1, 2 or 3) or external IO indicated by 0. The inputSwitch names can be defined on the configuration screen. The status could be 0 (for inputSwitch going low/off) or 1 (for inputSwitch going high/on). If timeOutSeconds is not specified or is negative value, the program will wait indefinitely (default).

Example Following examples show how to wait for the input switch status for different devices. In the first example the command waits at most 10 seconds for "MotorX_Input_Switch" located on motorX to go high. In the second example the command waits indefinitely for "External_IO_Input_Switch" located on external IO unit (optional) to go low.

```
                sla.WaitForSwitch(1, "MotorX_Input_Switch", 1,
10)

                sla.WaitForSwitch(0, "External_IO_Input_Switch",
0)
```

Group SLA I/O Commands

EXCEL Commands Reference

Following are the EXCEL Commands which can be used in a Multi Motor Program (MMP). They are arranged in related groups and in alphabetic order for easy access. All the EXCEL commands can be used in a MMP program by using "excel." namespace (type CTRL-SPACE after typing excel. in the MMP Programming environment.)

EXCEL Commands in Related Groups

The EXCEL Commands are categorized in each group depending on their functionalities.

- **EXCEL Initialization Commands**

  This command is used to initialize internal Excel cache.

  - ReadFile

- **EXCEL Interaction Commands**

  This command is used to read the values from the internal Excel cache.

  - GetValue

EXCEL Commands in Alphabetic Order

1. GetValue
2. ReadFile

EXCEL Commands

1. ReadFile

Summary Read from the Excel file.

Declaration ReadFile(fileName As String, Optional sheetName As String)

Return Returns True or False depending on the success or failure of the command.

Description ReadFile command read from the specified Excel file and optionally specified sheet (default is 'Sheet1'). If read attempt fails (e.g. file is not present), it returns False

Example Following is an example of reading from C:\Data.xls file's 'Main' sheet.

```
Dim result as Boolean

result = excel.ReadFile("C:\Data.xls", "Main")
```

Group EXCEL Initialization Commands

2. GetValue

Summary Get the value from Excel spreadsheet.

Declaration GetValue(row As Integer, col As String)

Return Returns a string value.

Description GetValue command gets the value of row and column location from the earlier read Excel spreadsheet. The column argument can be specified as either number or column letter. All values are returned as string which can be converted to the appropriate type..

Example Following is an example of reading a value for row 2, column A and assigning it to result Integer variable. Note that the same value can be obtained by using column number 1.

```
Dim result as Integer

result = CInt(excel.GetValue(2,"A"))

result = CInt(excel.GetValue(2,"1")) 'same value
```

Group EXCEL Interaction Commands

VISION Commands Reference

Following are the VISION Commands which can be used in a Multi Motor Program (MMP). They are arranged in related groups and in alphabetic order for easy access. All the VISION commands can be used in a MMP program by using "vision." namespace (type CTRL-SPACE after typing vision. in the MMP Programming environment.)

VISION Commands in Related Groups

The VISION Commands are categorized in each group depending on their functionalities.

- **VISION Connection Commands**

  These commands are used to establish connection to vision system.

  - ConnectCognex
  - ConnectDVT

- **VISION Interaction Commands**

  These commands are used to interact with the vision system.

  - GetCognexValue
  - GetDVTValue
  - ProcessCognex
  - ProcessDVT

VISION Commands in Alphabetic Order

1. ConnectCognex
2. ConnectDVT
3. GetCognexValue
4. GetDVTValue
5. ProcessCognex
6. ProcessDVT

VISION Commands

1. ConnectCognex

Summary Connect to the Cognex vision system.

Declaration ConnectCognex(host As String, port as Integer, Optional user As String, Optional password As String)

Return Returns True or False depending on the success or failure of the command.

Description Connect command establishes connection to the Cognex vision system on host and port with optional user and password. If connection fails (e.g. user or password is wrong), it returns False

Example Following is an example of connecting to Cognex vision system on host 'localhost', port 23 and user and password as 'admin'.

```
Dim result as Boolean

result =
vision.ConnectCognex("localhost",23,"admin","admin")
```

Group VISION Connection Commands

2. ConnectDVT

Summary Connect to the DVT vision system.

Declaration ConnectDVT(host As String, port as Integer, Optional user As String, Optional password As String)

Return Returns True or False depending on the success or failure of the command.

Description Connect command establishes connection to the DVT vision system on host and port with optional user and password. If connection fails (e.g. user or password is wrong), it returns False

Example Following is an example of connecting to DVT vision system on host 'localhost', port 5000.

```
Dim result as Boolean

result = vision.ConnectDVT("localhost",5000)
```

Group VISION Connection Commands

3. GetCognexValue

Summary Read the value from Cognex Vision System.

Declaration GetCognexValue(col As String, row As Integer)

Return Returns a string value.

Description GetCognexValue command reads the value from col and row location on Cognex system spreadsheet. In Cognex system, each column is alphabetically named (a, b, c etc.) and each row is numerically numbered (1, 2, 3 etc.). All values are returned as text.

Example Following is an example of reading a value for spreadsheet location a, 2 and assigning it to result Integer variable.

```
Dim result as Integer

result = CInt(vision.GetCognexValue("a",2))
```

Group VISION Interaction Commands

4. GetDVTValue

Summary Read the value from DVT Vision System.

Declaration GetDVTValue(key As String)

Return Returns a string value.

Description GetDVTValue command returns the value represented by key as string.

Example Following is an example of reading a value associated with key 'radius' and assigning it to result Integer variable.

```
Dim result as Integer

result = CInt(vision.GetDVTValue("radius"))
```

Group VISION Interaction Commands

5. ProcessCognex

Summary Process the image capture.

Declaration ProcessCognex()

Return Returns True or False depending on the success or failure of the command.

Description Process command triggers the image capture and subsequent processing of the acquired image.

Example Following is an example of processing once the part is in place.

```
vision.ProcessCognex()
```

Group VISION Interaction Commands

6. ProcessDVT

Summary Process the image capture.

Declaration ProcessDVT()

Return Returns True or False depending on the success or failure of the command.

Description ProcessDVT command triggers the image capture and subsequent processing of the acquired image.

Example Following is an example of processing once the part is in place.

```
vision.ProcessDVT()
```

Group VISION Interaction Commands

HMI Commands Reference

Following are the HMI Commands which can be used in a Multi Motor Program (MMP). They are arranged in related groups and in alphabetic order for easy access. All the HMI commands can be used in a MMP program by using "hmi." namespace (type CTRL-SPACE after typing hmi. in the MMP Programming environment.)

HMI Commands in Related Groups

The HMI Commands are categorized in each group depending on their functionalities.

- **HMI Data Commands**

  These commands are used to control the I/O switches. They can be used to control the integrated I/O as well as externally supplied unit.

  - ReadBinary
  - ReadRegister
  - SaveIO
  - WriteBinary
  - WriteRegister

- **HMI Motion Commands**

  These commands directly related to individual motors that drive the slides. They can be used to obtain useful status information regarding motors.

  - WaitForStop

HMI Commands in Alphabetic Order

1. ReadBinary
2. ReadRegister
3. SaveIO
4. WaitForStop
5. WriteBinary
6. WriteRegister

HMI Commands

1. ReadBinary

Summary Read a binary number from the HMI unit.

Declaration ReadBinary(ioNumber As Integer)

Return Returns a binary Integer value.

Description ReadBinary command reads a binary (0 or 1) value from the Master HMI unit for the specified ioNumber.

Example Following is an example for reading a binary value for ioNumber 3 and assigning it to ioValue Integer variable.

```
Dim ioValue as Integer

ioValue = hmi.ReadBinary(3)
```

Group HMI Data Commands

2. ReadRegister

Summary Read an unsigned 16 bit integer from the HMI unit.

Declaration ReadRegister(ioNumber As Integer)

Return Returns a integer Long value.

Description ReadRegister command reads an unsigned 16 bit integer (0 to 65535) value from the Master HMI unit for the specified ioNumber.

Example Following is an example for reading a value for ioNumber 3 and assigning it to ioValue Long variable.

```
Dim ioValue as Long

ioValue = hmi.ReadRegister(3)
```

Group HMI Data Commands

3. SaveIO

Summary Save Modbus IO data to the persistent storage.

Declaration SaveIO( )

Return No return value.

Description SaveIO command writes the current values of Modbus registers and coils to the persist ant storage. These values are restored when the program starts again.

Example Following is an example for saving the Modbus IO data.

```
hmi.SaveIO()
```

Group HMI Data Commands

4. WaitForStop

Summary Wait for the motion to be completed.

Declaration WaitForStop(Optional motorIndex As Integer, Optional monitorIONumber As Integer, Optional monitorIOStatus As Integer, Optional deaccelerate As Boolean)

Return No return value.

Description WaitForStop command waits for the end of motion of all motors (default). If the optional motorIndex argument is specified then it waits for the end of motion only for that motor. The motorIndex parameter could be any of the motors (1, 2 or 3). The additional two optional arguments can be used together to provide a monitor signal which when activated (has the value specified by monitorIOStatus) stops the motion. The value for monitorIONumber is any binary ioNumber on the HMI unit. monitorInputSwitchStatus indicates on what status (0 or 1) the motion stops. The optional parameter deaccelerate (True by default), controls the velocity profile while stopping the motor. If True then the motor deaccelerates to stop, otherwise it stops immediately. This command blocks the program execution till either the motion stops (if no monitor related arguments are specified) or when the monitor signal is activated (if monitor related arguments are specified). This command can be used along with sla.DoRelativeMove, sla.DoPositionMove or sla.DoVelocityMove commands to achieve synchronized motion (as opposed to coordinated motion where each of the motors move in lockstep manner) where each motor runs independently to obtain highly optimized timing profile.

Example Following example shows starting synchronized motion by using sla.DoPositionMove and blocking till the signal at ioNumber 5 goes low (0) or the motion is completed on all motors. If the IO 5 is activated before the motion ends, the motors will come to abrupt and immediate stop, since the parameter deaccelerate is False.

```
sla.DoPositionMove 1, 100, 100, 1000
sla.DoPositionMove 2, 200, 200, 1000
sla.DoPositionMove 3, 500, 500, 1000
hmi.WaitForStop(0, 5, 0, False)
```

Group HMI Motion Commands

5. WriteBinary

Summary Write a binary number to the HMI unit.

Declaration WriteBinary(ioNumber As Integer, value As Integer)

Return No return value.

Description WriteBinary command writes a binary (0 or 1) value to the Master HMI unit for the specified ioNumber.

Example Following is an example for writing a binary value 1 to ioNumber 3.

```
hmi.WriteBinary(3, 1)
```

Group HMI Data Commands

6. WriteRegister

Summary Write an unsigned 16 bit integer to the HMI unit.

Declaration WriteRegister(ioNumber As Integer, value As Long)

Return No return value.

Description WriteRegister command writes an unsigned 16 bit integer (0 to 65535) value to the Master HMI unit for the specified ioNumber.

Example Following is an example for writing an integer value 314 to ioNumber 3.

```
hmi.WriteRegister(3, 314)
```

Group HMI Data Commands

SERIAL Commands Reference

Following are the SERIAL Commands which can be used in a Multi Motor Program (MMP). They are arranged in related groups and in alphabetic order for easy access. All the SERIAL commands can be used in a MMP program by using "serial." namespace (type CTRL-SPACE after typing serial. in the MMP Programming environment.)

SERIAL Commands in Related Groups

The SERIAL Commands are categorized in each group depending on their functionalities.

- **SERIAL Connection Commands**

  These commands are used to open and close a connection to the serial system.

  - ClosePort
  - OpenPort

- **SERIAL Interaction Commands**

  These commands are used to interact with the serial system.

  - ReadPort
  - WritePort

SERIAL Commands in Alphabetic Order

1. ClosePort
2. OpenPort
3. ReadPort
4. WritePort

SERIAL Commands

1. ClosePort

Summary Close previously opened serial port.

Declaration ClosePort()

Return No return value.

Description ClosePort command closes previously opened serial port. If the port is not open, this command has no effect.

Example Following is an example of closing an already opened serial port.

```
serial.ClosePort()
```

Group SERIAL Connection Commands

2. OpenPort

Summary Open specified port to the serial system.

Declaration OpenPort(portNum As Integer, Optional baudRate As Long, Optional parity As String, Optional dataBits As Integer, Optional stopBit As Double, Optional handshake As Integer)

Return No return value.

Description OpenPort command establishes connection to the serial system at the specified portNum (1 = COM1, 2 = COM2 etc.). The additional optional parameters can be specified to further control the communication. Valid and default values are shown below.

- baudRate - 110, 300, 600, 1200, 2400, 9600 (default), 14400, 19200, 28800, 38400, 56000, 128000, 256000
- parity - "E", "M", "N" (default), "O", "S"
- dataBits - 4, 5, 6, 7, 8 (default)
- stopBit - 1 (default), 1.5, 2
- handshake - 0 (none), 1 (XON/XOFF, default), 2 (RTS/CTS), 3 (both)

Example Following is an example of opening COM1 port with 19200 baudrate and using other default values.

```
serial.OpenPort 1, 19200
```

Group SERIAL Connection Commands

3. ReadPort

Summary Read from the previously opened serial port.

Declaration ReadPort()

Return Returns a string value.

Description ReadPort command reads from the previously opened port and returns a string value.

Example Following is an example of reading the serial port.

```
Dim response as String

response = serial.ReadPort()
```

Group SERIAL Interaction Commands

4. WritePort

Summary Write to the previously opened serial port.

Declaration WritePort(data As String)

Return No return value.

Description WritePort command writes data to the previously opened port.

Example Following is an example of writing to the serial port.

```
serial.WritePort("0RSP")
```

Group SERIAL Interaction Commands

SLA SMP Commands Reference

Following are the SMP Commands which can be used in a Smart Motor Program (SMP). They are arranged in related groups and in alphabetic order for easy access.

SMP Commands in Related Groups

The SMP Commands are categorized in each group depending on their functionalities.

- **SMP Status Commands**

  These commands can be used to obtain the binary status bits (B*) to use in a program or for reporting as the host commands (RB*).

  - ADDR
  - Ba
  - Bb
  - Bc
  - Bd
  - Be
  - Bf
  - Bh
  - Bi
  - Bk
  - Bl
  - Bm
  - Bo
  - Bp
  - Br
  - Bs
  - Bt
  - Bu
  - Bv
  - Bw
  - Bx
  - RBa
  - RBb
  - RBc
  - RBd
  - RBe
  - RBf
  - RBh
  - RBi
  - RBk
  - RBl
  - RBm
  - RBo
  - RBp
  - RBr
  - RBs

- RBt
- RBu
- RBw
- RBx
- RPW
- RS
- RW

SMP IO Commands

These commands are used to control the integrated I/O (12 input and 6 output) switches.

- I1 to I12
- O1 to O6

SMP Brake Commands (where optional brake exists)

Many SmartMotors are available with power safe brakes. These brakes will apply a force to keep the shaft from rotating should the SmartMotor lose power. The brakes used in SmartMotors are zero-backlash devices with extremely long life spans. It is well within their capabilities to operate interactively within an application. Care should be taken not to create a situation where the brake will be set repeatedly during motion. That will reduce the brake's life

- BRKENG
- BRKRLS
- BRKSRV
- BRKTRJ

SMP Tuning Commands

These (K*) commands help tune a SmartMotor.

- F
- KA=expression
- KD=expression
- KG=expression
- KI=expression
- KL=expression
- KP=expression
- KS=expression
- KV=expression

- ## SMP Reset Commands

  These commands help to reset system state flags.

  - Z
  - Za
  - Zb
  - Zc
  - Zd
  - Zf
  - Zl
  - Zr
  - Zs
  - Zu
  - Zw
  - ZS

- ## SMP Programming Commands

  Program commands are like chores, whether it is to turn on an output, set a velocity or start a move. A program is a list of these chores. When a programmed SmartMotor is powered-up or its program is reset with the Z command, it will execute its program from top to bottom, with or without a host P.C. Connected. This section covers the commands that control the program itself. Once the program is running, there are a variety of commands that can redirect program flow and most of those can do so based on certain conditions. How these conditional decisions are setup determines what the programmed SmartMotor will do, and exactly how "smart" it will actually be.

  - BREAK
  - C#
  - CASE
  - DEFAULT
  - ELSE
  - ELSEIF
  - END
  - ENDS
  - ENDIF
  - GOSUB#
  - GOTO#
  - IF
  - LOOP
  - RETURN
  - RUN
  - RUN?
  - SWITCH
  - TWAIT
  - WAIT
  - WHILE

- ## SMP Motion Commands

These commands are responsible for the motion of a SmartMotor in various modes.

- A
- D
- G
- MP
- MV
- P
- RA
- RD
- RP
- RV
- S
- V
- X

- ## SMP Variables Commands

Variables are data holders that can be read, set and changed within the program or over the communication channel.
A variable can be set to an expression with only one operator and two operands. The operators can be any of the following:

+ Addition
- Subtraction
* Multiplication
/ Division
& Bit wise AND
| Bit wise OR

The following are legal:
a=b+c, a=b+3, a=5+8
a=b-c, a=5-c, a=b-10
a=b*c, a=3*5, a=c*3
a=b/c, a=b/2, a=5/b
a=b&c, a=b&8
a=b|c, a=b|15

An array variable is one that has a numeric index component that allows the numeric selection of which variable a program is to access. Common use of the array variable type is to set up what is called a buffer. In many applications, the SmartMotor will be tasked with inputting data about an array of objects and to do processing on that data in the same order, but not necessarily at the same time. Under those circumstances it may be necessary to "buffer" or "store" that data while the SmartMotor processes it at the proper times.

To set up a buffer the programmer would allocate a block of memory to it, assign a variable to an input pointer and another to an output pointer. Both pointers would start out as zero and every time data was put into the buffer the input pointer would increment. Every time the data was used, the output buffer would likewise increment. Every time one of the pointers is incriminated, it would be checked for exceeding the allocated memory space and rolled back to zero in that event, where it would continue to increment as data came in. This is a first-in, first-out or "FIFO" circular

buffer. Be sure there is enough memory allocated so that the input pointer never overruns the output pointer.

In addition there is 32K of non-volatile EEPROM memory to store variables when they need to survive the motor powering down which can be accessed using EPTR, VST and VLD commands.

- @P
- @PE
- @V
- a...z
- aa...zzz
- al[index]
- aw[index]
- ab[index]
- EPTR=expression
- Ra ... Rz
- Raa ... Rzz
- Raaa ... Rzzz
- Rab[index]
- Ral[index]
- Raw[index]
- VLD
- VST

SMP Motor Commands

These commands set important characteristics of the SmartMotor.

- E=expression
- F=expression
- I
- O
- OFF
- RE
- RI
- RPE

SMP Debug Commands

These commands are used for aiding in debugging the SMP programs.

- PRINT
- SILENT
- TALK

SMP Commands in Alphabetic Order

SMP Commands

[note: *In the command description shown below, # is an Integer number*]

1. @P

Summary Current position

Declaration @P

Return Current position in smart motor unit.

Description This is actual current position.

Group SMP Status Commands

2. @PE

Summary Current position error

Declaration @PE

Return Current position error in smart motor unit.

Description This is actual current position error.

Group SMP Status Commands

3. @V

Summary Current velocity

Declaration @V

Return Current velocity in smart motor unit.

Description This is actual current speed of the motor.

Group SMP Status Commands

4. a...z

Summary User variables

Declaration None

Return No return value.

Description The first 26 variables are long integers (32 bits) and are accessed with the lower case letters of the alphabet, a, b, c, . . . x, y, z.
a=# Set variable a to a numerical value
a=exp Set variable a to value of an expression

Group SMP Status Commands

5. aa...zzz

Summary More user variables

Declaration None

Return No return value.

Description In addition to the first 26, there are 52 more long integer variables accessible with double and triple lower case letters: aa, bb, cc,. . . xxx, yyy, zzz. The memory space that holds these 52 variables is more flexible, however. This same variable space can be accessed with an array variable type. This memory space is further made flexible by the fact that it can hold 51 thirty two bit integers, or 101 sixteen bit integers, or 201 eight bit integers (all signed).

Group SMP Status Commands

6. al[index]

Summary Array variable 32 bit

Declaration al[i]=exp

Return No return value.

Description Set variable to a signed 32 bit value where index i = 0...50. The index i may be a number, a variable a thorough z, or the sum or difference of any two variables a thorough z (variables only).

Group SMP Status Commands

7. aw[index]

Summary Array variable 16 bit

Declaration aw[i]=exp

Return No return value.

Description Set variable to a signed 16 bit value where index i = 0...100. The index i may be a number, a variable a thorough z, or the sum or difference of any two variables a thorough z (variables only).

Group SMP Status Commands

8. ab[index]

Summary Array variable 8 bit

Declaration ab[i]=exp

Return No return value.

Description Set variable to a signed 8 bit value where index i = 0...200. The index i may be a number, a variable a thorough z, or the sum or difference of any two variables a thorough z (variables only).

Group SMP Status Commands

9. A

Summary Set absolute acceleration

Declaration A=expression

Return No return value.

Description Acceleration must be a positive integer within the range of 0 to 2,147,483,648. The default is zero forcing something to be entered to get motion. A typical value is 100. If left unchanged, while the motor is moving, this value will not only determine acceleration but also deceleration which will form a triangular or trapezoidal velocity motion profile. This value can be changed at any time. The value set does not get acted upon until the next G command is sent.

Group SMP Motion Commands

10. ADDR

Summary Motor's self address variable

Declaration

Return No return value.

Description

Group SMP Status Commands

11. Ba

Summary Over current status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

12. Bb

Summary Parity error status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

13. Bc

Summary Communication overflow status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

14. Bd

Summary Math overflow status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

15. Be

Summary Excessive position error status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

16. Bf

Summary Communications framing error status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

17. Bh

Summary Excessive temperature status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

18. Bi

Summary Index captured status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

19. Bk

Summary EEPROM data integrity status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

20. Bl

Summary Historical left limit status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

21. Bm

Summary Real time left limit status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

22. Bo

Summary Motor off status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

23. Bp

Summary Real time right limit status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

24. Br

Summary Historical right limit status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

25. Bs

Summary Syntax error status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

26. Bt

Summary Trajectory in progress status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

27. Bu

Summary Array index error status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

28. Bv

Summary EEPROM locked state (obsolete)

Declaration

Return No return value.

Description

Group SMP Status Commands

29. Bw

Summary Encoder wrap around status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

30. Bx

Summary Real time index inptut status bit

Declaration

Return No return value.

Description

Group SMP Status Commands

31. BRKENG

Summary Brake engage

Declaration BRKENG

Return No return value.

Description Issuing the BRKENG command will engate the brake.

Group SMP Brake Commands

32. BRKRLS

Summary Brake release

Declaration BRKRLS

Return No return value.

Description Issuing the BRKRLS command will release the brake.

Group SMP Brake Commands

33. BRKSRV

Summary Release brake when servo active, engage break when inactive

Declaration BRKSRV

Return No return value.

Description The command BRKSRV engages the brake automatically, should the motor stop servoing and holding position for any reason. This might be due to loss of power or just a position error, limit fault, over-temperature fault.

Group SMP Brake Commands

34. BRKTRJ

Summary Release brake when running a trajectory, engage under all other conditions. Turns servo off when the brake is engaged.

Declaration BRKTRJ

Return No return value.

Description The command BRKTRJ will engage the brake in response to all of the events described for command BRKSRV, plus any time the motor is not performing a trajectory. In this mode the motor will be off, and the brake will be holding it in position, perfectly still, rather than the motor servoing when it is at rest. As soon as another trajectory is started, the brake will release. The time it takes for the brake to engage and release is on the order of only a few milliseconds.

Group SMP Brake Commands

35. D

Summary Set relative distance

Declaration D=expression

Return No return value.

DescriptionThe D command allows a relative distance to be specified, instead of an absolute position. The number following is encoder counts and can be positive or negative. The relative distance will be added to the current position, either during or after a move. It is added to the desired position rather than the actual position so as to avoid the accumulation of small errors due to the fact that any servo motor is seldom exactly where it should be at any instant in time.

Group SMP Motion Commands

36. E=expression

Summary Set allowable position error

Declaration

Return No return value.

Description

Group SMP Status Commands

37. END

Summary End program execution

Declaration END

Return No return value.

Description If it's necessary to stop a program, use an END command and execution will stop at that point. An END command can also be sent by the host to intervene and stop a program running within the motor. The SmartMotor program is never erased until a new program is downloaded. To erase the program in a SmartMotor, download only the END command as if it were a new program and that's the only command that will be left on the SmartMotor until a new program is downloaded. To compile properly, every program needs and END somewhere, even if it is never reached. If the program needs to run continuously, the END statement has to be outside the main loop

Group SMP Programming Commands

38. EPTR=expression

Summary Set data EEPROM pointer, 0-7999

Declaration EPTR=expression

Return No return value.

Description To read or write into this memory space it is necessary to properly locate the pointer. This is accomplished by setting EPTR equal to the offset.

Group SMP Status Commands

39. F

Summary Load filter

Declaration

Return No return value.

Description

Group SMP Status Commands

40. F=expression

Summary Special functions control

Declaration

Return No return value.

Description

Group SMP Status Commands

41. G

Summary Start motion (GO)

Declaration G

Return No return value.

Description The G command does more than just start motion. It can be used dynamically during motion to create elaborate profiles. Since the SmartMotor allows position, velocity and acceleration to change during motion, "on-the-fly", the G command can be used to trigger the next profile at any time.
Note: G also resets several system state flags.

Group SMP Motion Commands

42. GOSUB#, RETURN

Summary Call a subroutine, Return from subroutine

Declaration

Return No return value.

Description Just like the GOTO# command, the GOSUB# command, in conjunction with a C# label, will redirect program execution to the location of the label. But, unlike the GOTO# command, the C# label needs a RETURN command to return the program execution to the location of the GOSUB# command that initiated the redirection. There may be many sections of a program that need to perform the same basic group of commands. By encapsulating these commands between a C# label and a RETURN, they may be called any time from anywhere with a GOSUB#, rather than being repeated in their totality over and over again. There can be as many as one thousand different subroutines (0 -999) and they can be accessed as many times as the application requires.
By pulling sections of code out of a main loop and encapsulating them into subroutines, the main code can also be easier to read. Organizing code into multiple subroutines is a good practice.
The commands that can conditionally direct program flow to different areas use a constant [#] like 1 or 25, a variable like a or al[#] or a function involving constants and/or variables a+b or a/[#]. Only one operator can be used in a function. The following is a list of the operators:

+ Addition
- Subtraction
* Multiplication
/ Division
== Equals (use two =)
!= Not equal
< Less than
> Greater than
<= Less than or equal
>= Greater than or equal
& Bit wise AND
| Bit wise OR

Group SMP Programming Commands

43. GOTO#, C#

Summary Redirect program flow, Subroutine label (C0-C999)

Declaration GOTO#, C#

Return No return value.

Description The most basic commands for redirecting program flow, without inherent conditions, are GOTO# in conjunction with C#. Labels are the letter C followed by a number (#) between 0

and 999 and are inserted in the program as place markers. If a label, C1 for example, is placed in a program and that same number is placed at the end of a GOTO command, GOTO1, the program flow will be redirected to label C1 and the program will proceed from there.
As many as a thousand labels can be used in a program (0 -999), but, the more GOTO commands used, the harder the code will be to debug or read. Try using only one and use it to create the infinite loop necessary to keep the program running indefinitely, as some embedded programs do. Put a C1 label near the beginning of the program, but after the initialization code and a GOTO1 at the end and every time the GOTO1 is reached the program will loop back to label C1 and start over from that point until the GOTO1 is reached, again, which will start the process at C1 again, and so on. This will make the program run continuously without ending. Any program can be written with only one GOTO. It might be a little harder, but it will tend to force better program organization, which in turn, will make it easier to be read and changed.
Note: Calling subroutines from the host can crash the stack.

Group SMP Programming Commands

44. I

Summary Hardware index position variable

Declaration

Return No return value.

Description

Group SMP Status Commands

45. I1 to I12

Summary Set/Get state of Input switch I1,I2...I12

Declaration

Return No return value.

Description

Group SMP Status Commands

46. IF, ELSE, ELSEIF, ENDIF

Summary Conditional test

Declaration IF expression, If structure element, If structure element, End IF statement

Return No return value.

Description Once the execution of the code reaches the IF command, the code between that IF and the following ENDIF will execute only when the condition directly following the IF command is true.

```
IF a == 1
```

```
    b = 1
ENDIF
```

Variable b will only get set to one if variable a is equal to one. If a is not equal to one, then the program will continue to execute using the command following the ENDIF command. Notice also that the SmartMotor language uses a single equal sign (=) to make an assignment, such as where variable a is set to equal the logical state of input A. Alternatively, a double equal (==) is used as a test, to query whether a is equal to 1 without making any change to a. These are two different functions. Having two different syntaxes has farther reaching benefits.
The ELSE and ELSEIF commands can be used to add flexibility to the IF statement. If it were necessary to execute different code for each possible state of variable a, the program could be written as follows:

```
IF a == 0
    b = 1
ELSEIF a == 1
    c = 1
ELSEIF a == 2
    c = 2
ELSE
    d = 1
ENDIF
```

There can be many ELSEIF statements, but at most one ELSE. If the ELSE is used, it needs to be the last statement in the structure before the ENDIF. There can also be IF structures inside IF structures. That's called "nesting" and there is no practical limit to the number of structures that can nest within one another.

Group SMP Programming Commands

47. KA=expression

Summary PID acceleration feed-forward

Declaration

Return No return value.

Description

Group SMP Status Commands

48. KD=expression

Summary PID derivative compensation

Declaration

Return No return value.

Description

Group SMP Status Commands

49. KG=expression

Summary PID gravity compensation

Declaration

Return No return value.

Description

Group SMP Status Commands

50. KI=expression

Summary PID integral compensation

Declaration

Return No return value.

Description

Group SMP Status Commands

51. KL=expression

Summary PID integral limit

Declaration

Return No return value.

Description

Group SMP Status Commands

52. KP=expression

Summary PID proportional compensation

Declaration

Return No return value.

Description

Group SMP Status Commands

53. KS=expression

Summary PID derivative term sample rate

Declaration

Return No return value.

Description

Group SMP Status Commands

54. KV=expression

Summary PID velocity feed forward

Declaration

Return No return value.

Description

Group SMP Status Commands

55. MP

Summary Enable position mode

Declaration MP

Return No return value.

Description Position mode is the default mode of operation for the SmartMotor. If the mode were to be changed, the MP command would put it back into position mode. In position mode, the P and D commands will govern motion

Group SMP Motion Commands

56. MV

Summary Enable velocity mode

Declaration MV

Return No return value.

Description Velocity mode will allow continuous rotation of the motor shaft. In Velocity mode, the programmed position using the P or the D commands is ignored. Acceleration and velocity need to be specified using the A= and the V= commands. After a G command is issued, the motor will accelerate up to the programmed velocity and continue at that velocity indefinitely. In velocity mode as in Position mode, Velocity and Acceleration are changeable on-the-fly, at any

time. Simply specify new values and enter another G command to trigger the change. In Velocity mode the velocity can be entered as a negative number, unlike in Position mode where the location of the target position determines velocity direction or sign. If the 32 bit register that holds position rolls over in velocity mode it will have no effect on the motion.

Group SMP Motion Commands

57. O

Summary Set/Reset origin to any position

Declaration O=expression

Return No return value.

Description The O command (using the letter O, not the number zero) allows the host or program not just to declare the current position zero, but to declare it to be any position, positive or negative. The exact position to be re-declared is the ideal position, not the actual position which may be changing slightly due to hunting or shaft loading. The O= command directly changes the motor's position register and can be used as a tool to avoid +/-31 bit roll over position mode problems. If the SmartMotor runs in one direction for a very long time it will reach position +/-2,147,483,648 which will cause the position counter to change sign. While that is not an issue with Velocity Mode, it can create problems in position mode

Group SMP Motor Commands

58. O1 to O6

Summary Set/Get state of Output switch O1,O2...O6

Declaration

Return No return value.

Description

Group SMP Status Commands

59. OFF

Summary Turn motor servo off

Declaration OFF

Return No return value.

Description The OFF command will stop the motor from servoing, much as a position\ error or limit fault would. When the servo is turned off, one of the status LEDs will revert from Green to Red.

Group SMP Motor Commands

60. P

Summary Set position

Declaration P=expression

Return No return value.

Description The P command sets an absolute end position. The units are encoder counts and can be positive or negative. The end position can be set or changed at any time during or at the end of previous moves.

Group SMP Motion Commands

61. PRINT

Summary Print to RS-232

Declaration PRINT()

Return No return value.

Description A variety of data formats can exist within the parentheses of the PRINT() command. A text string is marked as such by enclosing it between double quotation marks. Variables can be placed between the parentheses as well as two variables separated by one operator. To send out a specific byte value, prefix the value with the # sign and represent the value with as many as three decimal digits ranging from 0 to 255. e.g. it is necessary to send #13 as the last character while using SLA OS software's SMP debug mode. Multiple types of data can be sent in a single PRINT() statement by separating the entries with commas. Do not use spaces outside of text strings because SmartMotors use spaces as delimiters along with carriage returns and line feeds. The following are all valid print statements and will transmit data through the main RS-232 channel:

```
PRINT("Hello world",#13)  'text
PRINT(a*b,#13)            'expression
PRINT(#32,#13)            'data
PRINT("A",a,a*b,#13)      'all
```

Group SMP Debug Commands

62. Ra ... Rz

Summary Report variables

Declaration Ra ... Rz

Return Return long integer (32 bits) value of the variable.

Description Report variables a ... z

Group SMP Status Commands

63. Raa ... Rzz

Summary Report variables

Declaration Raa ... Rzz

Return Return long integer (32 bits) value of the variable.

Description Report variables aa ... zz

Group SMP Status Commands

64. Raaa ... Rzzz

Summary Report variables

Declaration Raaa ... Rzzz

Return Return long integer (32 bits) value of the variable.

Description Report variables aaa ... zzz

Group SMP Status Commands

65. Rab[index]

Summary Report byte array variables (8-bit)

Declaration Rab[index]

Return Return 8 bit variable value.

Description Report 8 bit variable value Rab[i]

Group SMP Status Commands

66. Ral[index]

Summary Report long array variables (32-bit)

Declaration Ral[index]

Return Return 32 bit variable value.

Description Report 32 bit variable value Ral[i]

Group SMP Status Commands

67. Raw[index]

Summary Report word array variables (16-bit)

Declaration Raw[index]

Return Return 16 bit variable value.

Description Report 16 bit variable value Raw[i]

Group SMP Status Commands

68. RA

Summary Report acceleration

Declaration RA

Return Return acceleration in smart motor unit.

Description Report buffered acceleration.

Group SMP Status Commands

69. RBa

Summary Report over current status

Declaration

Return No return value.

Description

Group SMP Status Commands

70. RBb

Summary Report parity error status

Declaration

Return No return value.

Description

Group SMP Status Commands

71. RBc

Summary Report communications error status

Declaration

Return No return value.

Description

Group SMP Status Commands

72. RBd

Summary Report user math overflow status

Declaration

Return No return value.

Description

Group SMP Status Commands

73. RBe

Summary Report position error status

Declaration

Return No return value.

Description

Group SMP Status Commands

74. RBf

Summary Report communications framing

Declaration

Return No return value.

Description

Group SMP Status Commands

75. RBh

Summary Report overheat status

Declaration

Return No return value.

Description

Group SMP Status Commands

76. RBi

Summary Report index status

Declaration

Return No return value.

Description

Group SMP Status Commands

77. RBk

Summary Report EEPROM read/write status

Declaration

Return No return value.

Description

Group SMP Status Commands

78. RBl

Summary Report historical left limit status

Declaration

Return No return value.

Description

Group SMP Status Commands

79. RBm

Summary Report negative limit status

Declaration

Return No return value.

Description

Group SMP Status Commands

80. RBo

Summary Report motor off status

Declaration

Return No return value.

Description

Group SMP Status Commands

81. RBp

Summary Report positive limit status

Declaration

Return No return value.

Description

Group SMP Status Commands

82. RBr

Summary Report historical right limit status

Declaration

Return No return value.

Description

Group SMP Status Commands

83. RBs

Summary Report program scan status

Declaration

Return No return value.

Description

Group SMP Status Commands

84. RBt

Summary Report trajectory status

Declaration

Return No return value.

Description

Group SMP Status Commands

85. RBu

Summary Report user array index status

Declaration

Return No return value.

Description

Group SMP Status Commands

86. RBw

Summary Report wrap around status

Declaration

Return No return value.

Description

Group SMP Status Commands

87. RBx

Summary Report hardware indexinput level

Declaration

Return No return value.

Description

Group SMP Status Commands

88. RD

Summary Return buffered move distance value

Declaration

Return No return value.

Description

Group SMP Status Commands

89. RE

Summary Report buffered maximum position error

Declaration

Return No return value.

Description

Group SMP Status Commands

90. RI

Summary Report last stored index position

Declaration

Return No return value.

Description

Group SMP Status Commands

91. RP

Summary Report present position

Declaration

Return No return value.

Description

Group SMP Status Commands

92. RPE

Summary Report present position error

Declaration

Return No return value.

Description

Group SMP Status Commands

93. RPW

Summary Report position and status

Declaration

Return No return value.

Description

Group SMP Status Commands

94. RS

Summary Report status byte

Declaration

Return No return value.

Description

Group SMP Status Commands

95. RUN

Summary Execute stored user program

Declaration RUN

Return No return value.

Description If the SmartMotor is reset with a Z command, all previous variables and mode changes will be erased for a fresh start and the program will begin to execute from the top. Alternatively the RUN command can be used to start the program, in which case the state of the motor is unchanged and its program will be invoked.

Group SMP Programming Commands

96. RUN?

Summary Halt program if no RUN issued

Declaration Run?

Return No return value.

Description To keep a downloaded program from executing at power-up start the program with the RUN? Command. It will prevent the program from starting when power is applied, but it will not prevent the program from running when the SmartMotor sees a RUN command from a host over the RS-232 port.

Group SMP Programming Commands

97. RV

Summary Report velocity

Declaration

Return No return value.

Description

Group SMP Status Commands

98. RW

Summary Report status word

Declaration

Return No return value.

Description

Group SMP Status Commands

99. S

Summary Stop move in progress abruptly

Declaration S

Return No return value.

Description If the S command is issued while a move is in progress it will cause an immediate and abrupt stop with all the force the motor has to offer. After the stop, assuming there is no position error, the motor will still be servoing. The S command works in both Position and Velocity modes.

Group SMP Motion Commands

100. SILENT, TALK

Summary Suppress/Enable PRINT() outputs

Declaration S

Return SILENT, TALK

Description SILENT command causes all PRINT() output to be suppressed. This is necessary when host controller is also sending the commands to the SmartMotors as it can interfere with the echo mechanism. To de-assert silent mode, issue TALK command. This is useful for the debugging of SMP programs along with the Motor terminal tab of the SLA OS software. To receive the PRINT statements from the SMP program, select the checkbox at the top of the Motor terminal to receive output in the window.

Group SMP Debug Commands

101. SWITCH, CASE, DEFAULT, BREAK, ENDS

Summary Program execution control, Switch-case structure element, Switch-case structure element, Program execution flow control, Program execution control

Declaration SWITCH expression, CASE expression, DEFAULT, BREAK, ENDS

Return No return value.

Description Long, drawn out IF structures can be cumbersome, and burden the program visually. In these instances it can be better to use the SWITCH structure. The following code would accomplish the same thing as the second example program given in the IF command:

```
SWITCH a
    CASE 0
       b=1
       BREAK
    CASE 1
       c=1
       BREAK
    CASE 2
       c=2
```

```
        BREAK
    DEFAULT
        d=1
        BREAK
ENDS
```

Just as a rotary switch directs electricity, the SWITCH structure directs the flow of the program. The BREAK statement then jumps the code execution to the code following the associated ENDS command. The DEFAULT command covers every condition other than those listed. It is optional. Note: The SWITCH statement makes use of the same memory space as variable "zzz". Do not use this variable or array space when using SWITCH.

Group SMP Programming Commands

102. TWAIT

Summary Wait during trajectory

Declaration TWAIT

Return No return value.

Description The TWAIT command pauses program execution while the motor is moving. Either the controlled end of a trajectory, or the abrupt end of a trajectory due to an error, will terminate the TWAIT waiting period. If there were a succession of move commands without this command, or similar waiting code between them, the commands would overtake each other because the program advances, even while moves are taking place. The following program has the same effect as the TWAIT command, but has the added virtue of allowing other things to be programmed during the wait, instead of just waiting. Such things would be inserted between the two commands.

```
WHILE Bt
LOOP
```

Group SMP Programming Commands

103. V

Summary Set maximum permitted velocity

Declaration V=expression

Return No return value.

Description Use the V command to set a limit on the velocity the motor can accelerate to. That limit becomes the slew rate for all trajectory based motion whether in position mode or velocity mode. The value defaults to zero so it must be set before any motion can take place. The new value does not take effect until the next G command is issued.

Group SMP Motion Commands

104. VLD

Summary Sequentially load variables from data EEPROM

Declaration VLD(variable,index)

Return No return value.

Description To load variables, starting at the pointer, use the VLD command. In the "variable" space of the command put the name of the variable and in the "index" space put the number of sequential variables to be loaded.

Group SMP Status Commands

105. VST

Summary Sequentially store variables to data EEPROM

Declaration VST(variable,index)

Return No return value.

Description To store a series of variables, use the VST command. In the "variable" space of the command put the name of the variable and in the "index" space put the total number of sequential variables that need to be stored. Enter a one if just the variable specified needs to be stored. The actual sizes of the variables will be recognized automatically.

Group SMP Status Commands

106. WAIT

Summary Wait (exp) sample periods

Declaration WAIT=exp

Return No return value.

Description There will probably be circumstances where the program execution needs to be paused for a specific period of time. Time, within the SmartMotor, is tracked in terms of servo sample periods. It is recommended that the Unit Conversion calculator available in the software be used for getting the corresponding number from seconds. The following code would be the same as WAIT=1000, only it will allow code to executed during the wait if it is placed between the WHILE and the LOOP.

```
CLK = 0           'Reset CLK to 0
WHILE CLK
```

Group SMP Programming Commands

107. WHILE, LOOP

Summary Conditional program flow command, While structure element

Declaration WHILE expression, LOOP

Return No return value.

Description The most basic looping function is a WHILE command. The WHILE is followed by an expression that determines whether the code between the WHILE and the following LOOP command will execute or be passed over. While the expression is true, the code will execute. An expression is true when it is non-zero. If the expression results in a "zero" then it is false. The following is an example of a valid WHILE loop which will execute ten times.

```
a = 1
WHILE a < 10
    a = a + 1
LOOP
```

The task or tasks within the WHILE loop will execute as long as the function remains true. The BREAK command can be used to break out of a WHILE loop, although that somewhat compromises the elegance of a WHILE statement's single test point, making the code a little harder to follow. The BREAK command should be used sparingly or preferably not at all in the context of a WHILE.

Group SMP Programming Commands

108. X

Summary Decelerate to stop

Declaration X

Return No return value.

Description If the X command is issued while a move is in progress it will cause the motor to decelerate to a stop at the last entered A= value. When the motor comes to rest it will servo in place until commanded to move again. The X command works in both Position and Velocity modes.

Group SMP Motion Commands

109. Z

Summary Total system reset

Declaration

Return No return value.

Description

Group SMP Status Commands

110. Za

Summary Reset current limit violation latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

111. Zb

Summary Reset serial data parity violation latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

112. Zc

Summary Reset communications buffer overflow latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

113. Zd

Summary Reset math overflow violation latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

114. Zf

Summary Reset serial comm framing error latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

115. Zl

Summary Reset historical left limit latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

116. Zr

Summary Reset historical right limit latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

117. Zs

Summary Reset command scan error latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

118. Zu

Summary Reset user array index access latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

119. Zw

Summary Reset encoder wrap around event latch bit

Declaration

Return No return value.

Description

Group SMP Status Commands

120. ZS

Summary Reset system latches to power-up state

Declaration

Return No return value.

Description

Group SMP Status Commands

eCylinder/eRotary SMP Commands Reference

Following are the SMP Commands which can be used in a Smart Motor Program (SMP). They are arranged in related groups and in alphabetic order for easy access.

SMP Commands in Related Groups

The SMP Commands are categorized in each group depending on their functionalities.

- **SMP Motion Commands**

  These commands are responsible for the motion of a SmartMotor in various modes.

  - ACC
  - al
  - MODE
  - VEL

SMP Commands in Alphabetic Order

1. ACC
2. al
3. MODE
4. VEL

SMP Commands

[note: *In the command description shown below, # is an Integer number*]

1. ACC

Summary Set absolute acceleration for the specified index position.

Declaration ACC[index]=expression

Return No return value.

Description Acceleration must be a positive integer > 0. The default is zero forcing something to be entered to get motion. A typical value is 100. If left unchanged, while the motor is moving, this value will not only determine acceleration but also deceleration which will form a triangular or trapezoidal velocity motion profile. This value can be changed at any time. The value set does not get acted upon until the next G command is sent.

Group SMP Motion Commands

2. al

Summary Set position for the specified index position.

Declaration al[index]=expression

Return No return value.

Description The command sets an absolute end position. The units are encoder counts and can be positive or negative. The end position can be set or changed at any time during or at the end of previous moves.

Group SMP Motion Commands

3. MODE

Summary Set mode for the specified index position.

Declaration MODE[index]=expression

Return No return value.

Description The command sets either absolute (default) or relative mode for the position. If a given position has absolute mode then the motor moves to the specified position. If a given position has relative mode then the motor moves relative to the current position by incremental distance equal to the specified position.

Group SMP Motion Commands

4. VEL

Summary Set maximum permitted velocity for the specified index position.

Declaration VEL[index]=expression

Return No return value.

Description Use the VEL command to set a limit on the velocity the motor can accelerate to. That limit becomes the slew rate for all trajectory based motion. The value defaults to zero so it must be set before any motion can take place. The new value does not take effect until the next G command is issued.

Group SMP Motion Commands